1 Esercizi su Analisi del Costo Computazionale

1. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo Mystery1(Int n):

Algorithm 1: Mystery1(Int n) $\rightarrow$ Int

```
if n ≤ 1 then
    return 123
else
    k ←Mystery2(n/2)+Mystery1(n/3)
    return k+Mystery1(n/3)

/* funzione ausiliaria                                    */
function Mystery2(Int m)
if m = 0 then
    return 321
else
    return 2×Mystery2(m/4)−Mystery2(m/4)
```

2. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo Mystery1(Int n):

Algorithm 2: Mystery1(Int n) $\rightarrow$ Int

```
if n ≤ 1 then
    return 32
else
    return Mystery2(n/2)+Mystery1(n/2)

function Mystery2(Int m) → Int
if m = 1 then
    return 2
else
    return 2 × Mystery2(m − 1)
```

3. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n):

Algorithm 3: MYSTERY(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 1
else
    Int v ← 0
    for i ← 1 to n do
        v ← v + i
    return v + Mystery2(n − 1)

function Mystery2(Int m) → Int
if m ≤ 1 then
    return 1
else
    Int w ← 0
    for j ← m downto 1 do
        w ← w + j
    return w + Mystery(m − 1)
```

4. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n):

Algorithm 4: MYSTERY(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 1
else
    Int v ← 0
    for i ← 1 to n do
        for j ← n downto 1 do
            v ← v + i − j
    return v + Mystery(n/2) − Mystery2(n)

function Mystery2(Int m) → Int
Int x ← Mystery(m/2)
return 2 × x − Mystery(m/2)
```

Cont.

5. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n):

Algorithm 5: MYSTERY(INT n) $\rightarrow$ INT

```
i ← 1
j ← 0
while i ≤ n do
    i ← i × 2
    j ← j + MYSTERY2(n/2) + MYSTERY2(n/2)
return j

function MYSTERY2(INT m) → INT
if m ≤ 1 then
    return 1
else
    j ← 1
    while j ≤ m do
        j ← j + 1
    return j + MYSTERY2(m/3) + MYSTERY2(m/3)
```

6. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n):

Algorithm 6: MYSTERY(INT n) $\rightarrow$ INT

```
if m ≤ 1 then
    return 1
else
    i ← 1
    j ← 0
    while i ≤ n do
        i ← i + 2
        j ← j + MYSTERY2(n/2)
    return j + MYSTERY(n/4) + MYSTERY(n/4) + MYSTERY(n/4)

function MYSTERY2(INT m) → INT
if m ≤ 1 then
    return 1
else
    j ← 1
    while j ≤ m do
        j ← j + 1
    return j + MYSTERY2(m/3) + MYSTERY2(m/3)
```

Cont.

7. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n):

Algorithm 7: MYSTERY(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 1
else
    Int v ← Mystery(n/2)
    for i ← 1 to n do
        v ← v + Mystery2(n)
    return v + Mystery(n/2)

function Mystery2(Int m) → Int
Int v ← 0
for j ← m downto 1 do
    v ← v + j
return v
```

8. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n):

Algorithm 8: MYSTERY(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 1
else
    Int v ← 0
    for i ← n downto 1 do
        v ← v + i
    return v + Mystery2(n)

function Mystery2(Int m) → Int
if m ≤ 1 then
    return 1
else
    Int w ← 0
    for j ← 1 to m do
        w ← w + j
    return w + Mystery(m − 1)
```

Cont.

9. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY1(INT n):

Algorithm 9: MYSTERY1(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 32
else
    return MYSTERY2(n)+MYSTERY1(n − 1)

function MYSTERY2(INT m) → INT
if m = 1 then
    return 2
else
    return ( 2 × MYSTERY2(m/2) ) + ( 3 × MYSTERY2(m/2) )
```

10. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT $A[1..n]$):

Algorithm 10: MYSTERY(INT $A[1..n]$) $\rightarrow$ INT

```
bh ← new BinaryHeap()
k ← 0
i ← 1
while i ≤ n do
    bh.insert(A[i])
    k ← k + 1
    i ← i × 2
for j ← 1 to (k-1) do
    bh.deleteMax()
return bh.findMax()
```

11. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT $A[1..n]$):

Algorithm 11: MYSTERY(INT $A[1..n]$)

```
A.heapify()
i ← 1
while i ≤ n do
    print(A. findMax())
    A.deleteMax()
    i ← i + i
```

Cont.

12. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n) assumendo implementazione della struttura UnionFind tramite quickFind:

Algorithm 12: MYSTERY(INT n) $\rightarrow$ INT

```
UNIONFIND uf ← new UNIONFIND()
for j ← 1 to n do
    uf.makeSet(j)
INT u ← 1
INT v ← n
while u ≤ n do
    uf.union(uf.find(u), uf.find(v))
    u ← u × 2
    v ← v − 1
return uf.find(1)
```

13. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n) assumendo implementazione della struttura UnionFind tramite quickUnion con euristica "by rank":

Algorithm 13: MYSTERY(INT n) $\rightarrow$ INT

```
UNIONFIND uf ← new UNIONFIND()
for j ← 1 to n do
    uf.makeSet(j)
INT u ← 1
INT v ← n
while u ≤ n do
    uf.union(uf.find(u), uf.find(v))
    u ← u + u
    v ← v/2
return uf.find(1)
```

Cont.

14. **Esercizio.** Calcolare il costo computazionale $T(n)$ del seguente algoritmo MYSTERY(INT n) assumendo implementazione della struttura UnionFind tramite QuickFind:

Algorithm 14: MYSTERY(INT n) $\rightarrow$ INT

```
UNIONFIND uf ← new UNIONFIND()
for j ← 1 to n do
    uf.makeSet(j)
INT u ← 1
INT v ← n
INT z, k
while u ≤ n do
    z ← uf.find(u)
    k ← uf.find(v)
    uf.union(z, k)
    u ← 4 × u
    v ← v/4
return uf.find(1)
```

Cont.

2 Esercizi su Strutture di Dati

1. **Esercizio.** Si consideri una tabella hash di dimensione $m = 7$ inizialmente vuota. Le collisioni sono gestite mediante indirizzamento aperto usando la seguente funzione hash $h(k)$:

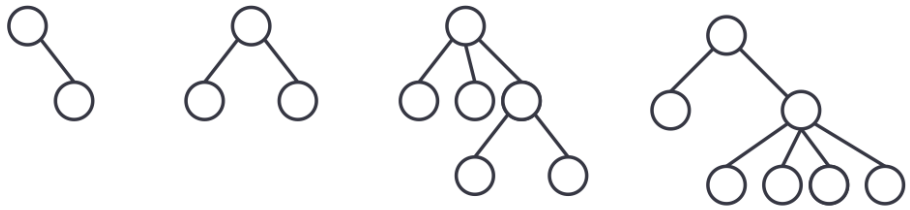
   $h(k) = (h'(k) + 3i) \mod m$

   $h'(k) = k \mod m$

   Si mostri il contenuto della tabella e il fattore di carico dopo ognuna delle seguenti operazioni, eseguite in ordine: INS(17), INS(4), INS(10), INS(25), INS(46), DEL(17), INS(11), DEL(25), INS(39).

2. **Esercizio.** Scrivere un algoritmo che prende in input due alberi binari T1 e T2 e restituisce *true* se T1 e T2 sono identici, cioè se hanno la stessa struttura e valori uguali in posizioni corrispondenti (radici uguali, figlio sinistro della radice di T1 uguale al figlio sinistro della radice di T2, figlio destro della radice di T1 uguale al figlio destro della radice di T2, e così via); l'algoritmo restituisce *false* in caso contrario.

3. **Esercizio.** Si scriva un algoritmo che prende in input un albero n-ario T e conta quanti sono i livelli che hanno un numero di nodi pari. Nei casi seguenti quindi restituisce rispettivamente 0, 1, 1, 2.

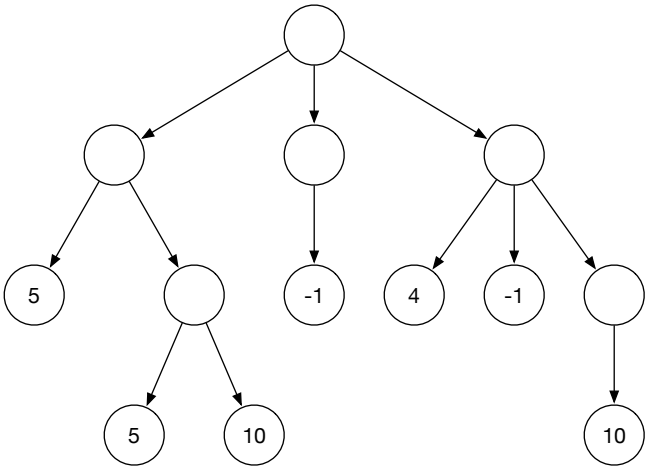


4. **Esercizio.** Scrivere un algoritmo che prende in input un albero binario T e un intero positivo k e calcola il numero di nodi che si trovano esattamente a profondità k (nota: la radice si trova a profondità zero). Discutere il costo computazionale nel caso pessimo e ottimo.

5. **Esercizio.** Scrivere un algoritmo che, preso in input un albero binario T ed un intero positivo k, ritorni il numero di foglie a profondità maggiore o uguale di k. Discutere il costo computazionale nel caso pessimo e nel caso ottimo. Nota: la radice si trova a profondità 0.

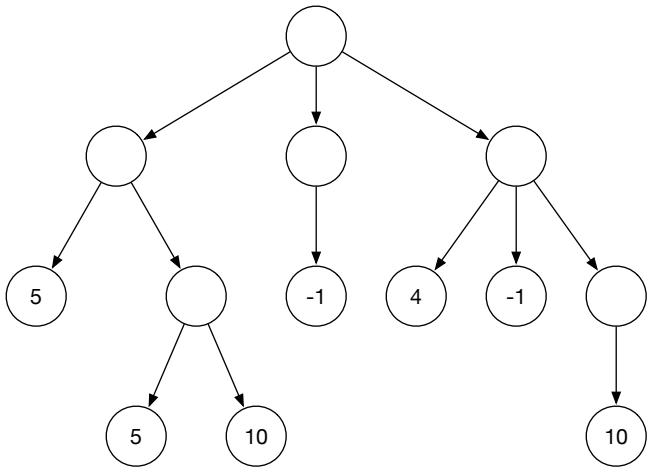
6. **Esercizio.** Si scriva una procedura che, date due liste concatenate monodirezionali, L_1 e L_2, contenenti interi **ordinati** dal più piccolo al più grande, rimuova da L_1 tutti gli interi che appaiono in L_2 (senza modificare L_2). Esempio: se L_1 = [1,1, 2, 3,4,4] e L_2 = [1,3,3,5], al termine della procedura L_1 = [2,4,4] e L_2 = [1,3,3,5]. Discutere il costo computazionale della procedura proposta nel caso ottimo e nel caso pessimo.

7. **Esercizio.** Dato il seguente albero, con label numeriche assegnate alle foglie, mostrare le label dei nodi intermedi dopo l'applicazione degli algoritmi Minimax e AlphaBeta pruning assumendo che la radice dell'albero sia un nodo di **massimizzazione** (i.e. prende il valore massimo tra i valori assegnati ai figli diretti). Nel disegnare l'albero dopo l'applicazione dell'AlphaBeta pruning non mostrare i rami dell'albero non visitati.



Cont.

8. **Esercizio.** Dato il seguente albero, con label numeriche assegnate alle foglie, mostrare le label dei nodi intermedi dopo l'applicazione degli algoritmi Minimax e AlphaBeta pruning assumendo che la radice dell'albero sia un nodo di **minimizzazione** (i.e. prende il valore minimo tra i valori assegnati ai figli diretti). Nel disegnare l'albero dopo l'applicazione dell'AlphaBeta pruning non mostrare i rami dell'albero non visitati.



9. **Esercizio.** Si scriva un algoritmo che preso in input un **albero binario di ricerca** T contenente chiavi intere non ripetute ed un intero $k \geq 1$, ritorni il k-esimo intero più piccolo contenuto nell'albero. Se T contiene meno di k chiavi, l'algoritmo ritorna NA (costante che indica valore non disponibile). Discutere il costo computazionale della soluzione proposta.

10. **Esercizio.** Si scriva un algoritmo che preso in input un **albero binario** T ritorni TRUE se T è *perfetto* e FALSE in caso contrario. Discutere il costo computazionale della soluzione proposta. Nota: un albero binario è *perfetto* se 1) ogni nodo che non sia una foglia ha esattamente due figli; 2) tutte le foglie sono alla stessa profondità.

11. **Esercizio.** Dato un albero AVL inizialmente vuoto effettuare le seguenti operazioni in ordine e mostrare lo stato dell'albero dopo ogni operazione:

    (a) insert 2 (b) insert 3 (c) insert 4 (d) insert 7 (e) delete 3
    (f) delete 2 (g) insert 1 (h) insert 8 (i) insert 5 (j) insert 6

    Indicare chiaramente quale operazione viene eseguita.

12. **Esercizio.** Dato un albero AVL inizialmente vuoto effettuare le seguenti operazioni in ordine e mostrare lo stato dell'albero dopo ogni operazione:

    (a) insert 4 (b) insert 2 (c) insert 1 (d) insert 6 (e) insert 3
    (f) insert 5 (g) insert 8 (h) insert 7 (i) delete 3 (j) delete 5

    Indicare chiaramente quale operazione viene eseguita.

13. **Esercizio.** Scrivere un algoritmo che, data in input una lista concatenata monodirezionale, rimuova l'elemento nel mezzo della lista senza effettuare due scansioni della lista. Se la lista ha un numero pari n di elementi, è possibile scegliere di rimuovere o l'elemento $n/2$ oppure l'elemento $n/2 + 1$.

14. **Esercizio.** Dato un albero AVL inizialmente vuoto effettuare le seguenti operazioni in ordine e mostrare lo stato dell'albero dopo ogni operazione:

    (a) insert 2 (b) insert 5 (c) insert 8 (d) insert 3
    (e) insert 1 (f) insert 4 (g) insert 6 (h) insert 7

    Indicare chiaramente quale operazione viene eseguita.

Cont.

3 Esercizi su Tecniche Algoritmiche

1. **Esercizio.** Si consideri il seguente balletto medievale ballato da n maschi e n femmine. Inizialmente tutti i ballerini si collocano in file lungo una linea retta in modo tale che ognuno sia a distanza di un passo dai propri vicini. In una fase iniziale, le femmine, una alla volta, si muovono per raggiungere un maschio con cui formeranno una coppia per il resto del balletto. Le femmine si muovono a tempo, e ad ogni battuta della musica, la femmina che si sta muovendo fa un passo spostandosi a proprio piacimento verso destra oppure sinistra. Una volta raggiunto il compagno con cui formerà la coppia, alla battuta successiva un'altra femmina inizierà i propri spostamenti. Data una disposizione iniziale dei ballerini, bisogna capire quanto tempo sarà necessario per completare la formazione delle coppie. Il tempo di formazione di una coppia coincide con il numero complessivo di passi che devono effettuare le femmine per completare la formazione delle coppie. Progettare quindi un algoritmo che dato un vettore di booleani $B[1..2n]$ che indica le posizioni iniziali dei ballerini ($B[i] = \mathit{true}$ indica che in posizione i è presente una femmina, altrimenti se $B[i] = \mathit{false}$ in posizione i è presente un maschio), restituisce un intero che rappresenta la durata minima possibile per la formazione di tutte le n coppie (ovvero, il numero minimo complessivo di passi che le femmine devono fare per formare le coppie).

2. **Esercizio.** Prima di entrare in una sala cinematografica è possibile riempire un contenitore, a forma di bicchiere, con delle caramelle. Il bicchiere può contenere caramelle fino ad un peso complessivo rappresentato da un numero intero K. Le caramelle si prendono da n distributori di caramelle; l'i-esimo distributore, con $1 \leq i \leq n$, contiene caramelle di peso rappresentato da un numero intero $p[i]$. Per comodità, assumiamo che ogni distributore contenga una quantità illimitata di caramelle. Si vuole riempire il bicchiere esattamente per la sua capacità K, massimizzando il peso medio delle caramelle presa. Il peso medio è il peso complessivo delle caramelle prese, diviso il loro numero. Progettare un algoritmo che, data la capacità del bicchiere K, ed il vettore $p[1..n]$, restituisce un numero reale indicante il massimo peso medio di caramelle che si possono prendere, assumendo di riempire il bicchiere esattamente per la sua capacità K.

3. **Esercizio.** Progettare un algoritmo che dati due vettori di numeri $A[1..n]$ e $B[1..n]$ calcola quanti indici $i \in \{1, \ldots, n\}$ sono tali che $A[i]$ appare anche in B.

4. **Esercizio.** Una cisterna contiene rifiuti gassosi tossici da smaltire. Lo smaltimento si effettua riempiendo particolari contenitori, ognuno avente una propria capacità possibilmente differente dalle capacità degli altri contenitori. Quando si collega la cisterna ad un contenitore per lo smaltimento del gas, il trasferimento del gas prevede di riempire completamente il contenitore, a meno che la cisterna non si svuoti completamente prima di terminare il riempimento del contenitore. Bisogna distribuire il gas nel numero massimo di contenitori possibili. Progettare un algoritmo che riceve in input la quantità K di metri cubi di gas inizialmente nella cisterna, ed un vettore V di lunghezza n tale che $V[i]$ è la capacità in metri cubi del i-esimo contenitore (i contenitori disponibili sono n). Scrivere un algoritmo che restituisce un vettore O di lunghezza n tale che $O[i]$ è la quantità di gas che verrà smaltita nel i-esimo contenitore. Si ricordi che lo smaltimento deve utilizzare più contenitori possibili. Potete assumere che la sommatoria delle capacità dei contenitori sia superiore alla quantità di gas da smaltire.

5. **Esercizio.** Un ladro entra in un appartamento in cui si trovano n oggetti, ognuno con un proprio valore $v[i]$. Ha a disposizione k borse da riempire con gli oggetti da rubare, con il vincolo che **ogni borsa** può contenere **un solo** oggetto. Bisogna aiutare il ladro a calcolare il massimo valore possibile del bottino. Dovete quindi progettare un algoritmo che dati i valori n (numero oggetti), k (numero borse), e l'array $v[1..n]$ (valori degli oggetti), restituisce il valore complessivo massimo che è possibile ottenere selezionando k oggetti. Riportare il costo computazionale (in tempo) della soluzione proposta, nel caso pessimo, nel caso ottimo, e possibilmente anche nel caso medio.

Cont.

6. **Esercizio.** Bisogna preparare il trasporto di un oggetto fragile, che viene collocato in un cartone che deve poi essere riempito con pezzi di polistirolo per limitarne i possibili danni durante il trasporto. Il volume complessivo del polistirolo da inserire è un intero K. Esistono n diversi formati di pezzi di polistirolo, ognuno con un proprio volume $v[i]$, con $v[1..n]$ array di interi. Per ogni formato, sono disponibili una quantità arbitraria di pezzi da poter utilizzare. Per rendere più sicura la spedizione, si desidera massimizzare il numero di pezzi di polistirolo da inserire nel cartone. Progettare un algoritmo che dati K e $v[1..n]$ restituisce un array $x[1..n]$ che indica che, per ottenere il volume K massimizzando il numero di pezzi di polistirolo, si possono usare $x[i]$ pezzi del formato i-esimo. Nel caso in cui non sia possibile raggiungere il volume complessivo K con i formati di polistirolo disponibili, l'array x conterrà valori tutti uguali a 0 (ovvero, $x[i] = 0$ per tutti gli $i \in \{1, \ldots, n\}$).

7. **Esercizio.** Progettare un algoritmo che dato un vettore di numeri $V[1..n]$ stampa una sequenza di indici $j_1, j_2, \ldots, j_k$, con k più grande possibile, tale che $j_i < j_{i+1}$ e $V[j_i] \geq V[j_{i+1}]$, per ogni $i \in \{1, \ldots, k-1\}$.

8. **Esercizio.** Su un pallone aerostatico in difficoltà di volo si deve ridurre il peso complessivo di almeno un valore W (numero naturale), buttando fuori dall'abitacolo alcuni oggetti. Esistono n oggetti che si possono buttare, ognuno con un proprio peso (un numero naturale) ed un proprio valore (un numero non negativo). Si vuole trovare la combinazione di oggetti ottimale, ovvero un insieme di oggetti di peso complessivo superiore a W che minimizzi il valore complessivo. Progettare un algoritmo che dati i vettori $P[1..n]$ e $V[1..n]$ che rappresentano i pesi ed i valori degli n oggetti, ed il valore W di peso da ridurre, stampa un insieme ottimale di oggetti da buttare dal pallone aerostatico per ridurre il peso di almeno una quantità W, minimizzando il valore complessivo che viene gettato.

9. **Esercizio.** Un appassionato di televisione vuole guardare la quantità massima di trasmissioni nella medesima giornata. Una trasmissione televisiva è caratterizzata da un istante di inizio s ed un istante di fine e tale che $s < e$. Due diverse trasmissioni con, rispettivamente, istanti di inizio-fine $[s_1, e_1]$ e $[s_2, e_2]$ possono essere viste entrambe solo se non si sovrappongono i relativi intervalli di inizio-fine, ovvero $e_1 < s_2$ oppure $e_2 < s_1$. Ad esempio, date tre trasmissioni televisive con istanti di inizio-fine, il numero massimo di trasmissioni che possono essere viste è due: prima la trasmissione con inizio-fine $[2, 3]$, seguita dalla trasmissione con inizio-fine $[4, 7]$. Scrivere un algoritmo che dato un array di lunghezza n, che contiene coppie di numeri che identificano gli intervalli di inizio-fine di n possibili trasmissioni, restituisce il numero massimo di trasmissioni che è possibile vedere (ovvero, con intervalli di inizio-fine non sovrapposti).

10. **Esercizio.** Una cava contiene delle pietre, ognuna con un proprio peso. Bisogna consegnare ad un cliente un preciso peso complessivo di pietre, e quindi si deve capire se è possibile raggiungere esattamente tale peso selezionando alcune delle pietre disponibili. Per risolvere tale problema, progettare un algoritmo che, dati un array di numeri interi non negativi $A[1..n]$, ed un numero intero non negativo K, restituisce *true* se esiste un sottoinsieme dei numeri in A che sommati danno K. Ad esempio, dato $A = [10, 5, 4, 8, 21]$ e $K = 13$, restituisce *true* in quanto $5 + 8 = 13$.

11. **Esercizio.** Un trasportatore deve caricare il proprio camion scegliendo alcuni fra n pacchi. Il pacco i-esimo ha un peso p_i ed un valore v_i (con $1 \leq i \leq n$). Si vuole caricare il camion con il peso complessivo massimo, ma per questioni di assicurazione il valore complessivo del carico non deve superare un dato valore K. Progettare un algoritmo che dati due vettori di interi P e V (di lunghezza n, tali che $P[i]$ e $V[i]$ contengono il peso p_i ed il valore v_i dell'i-esimo pacco) ed un intero K, stampi gli indici dei pacchi che costituiscono un carico, di valore complessivo minore o uguale a K, che massimizzi il peso complessivo.

12. **Esercizio.** Progettare un algoritmo che dato un vettore $V[1..n]$ contenente n numeri tutti differenti fra di loro, ed un intero K, tale che $0 \leq K \leq n-1$, restituisce quel valore x presente in V tale che V contiene esattamente K valori più piccoli di x e $(n - K - 1)$ valori più grandi di x. Discutere il costo dell'algoritmo nel caso ottimo, nel caso pessimo, e possibilmente anche nel caso medio.

13. **Esercizio.** Al fine di aumentare il numero di bambini vaccinati, un'azienda sanitaria decide di dedicare un infermiere a vaccinazioni da farsi direttamente presso le scuole. Esistono n scuole diverse, ognuna identificata da un numero naturale compreso fra 1 e n. L'infermiere ha a disposizione una quantità limitata H di tempo da dedicare alle vaccinazioni: andando presso la scuola i-esima, ha modo di vacccinare b_i bambini impiegando una quantità di tempo h_i. Progettare un algoritmo che dato il numero naturale H, ed i vettori di numeri naturali $b[1..n]$ e $h[1..n]$, che contengono rispettivamente i valori b_i e h_i, stampa gli indici delle scuole presso cui l'infermiere deve recarsi per massimizzare il numero complessivo di vaccinazioni.

Cont.

14. **Esercizio.** Si consideri una tabella bidimensionale di numeri $T[1..n, 1..m]$, con n righe ed m colonne, tale che i numeri letti colonna per colonna sono ordinati in senso non decrescente, ovvero, per ogni $i_1, i_2 \in \{1, \ldots, n\}$ e ogni $j_1, j_2 \in \{1, \ldots, m\}$ si ha che se $j_1 < j_2$ oppure ($j_1 = j_2$ e $i_1 \leq i_2$) allora $T[i_1, j_1] \leq T[i_2, j_2]$. Progettare un algoritmo che, data tale tabella T ed un numero x, verifica se x appare in T.

Cont.

4 Esercizi su Grafi

1. **Esercizio.** Progettare un algoritmo che, dato un grafo orientato $G = (V, E)$ ed un vertice $v \in V$, restituisce il numero di vertici di un ciclo di lunghezza minima a cui v appartiene (restituisce ∞ se v non appartiene ad alcun ciclo).

2. **Esercizio.** Progettare un algoritmo che, dato un grafo orientato pesato $G = (V, E, w)$, con tutti i pesi negativi (ovvero, per ogni coppia di vertici adiacenti u_1 e u_2 si ha $w(u_1, u_2) < 0$), e due vertici $s, t \in V$, restituisce, se esiste, il cammino da s a t di peso complessivo massimo.

3. **Esercizio.** Si consideri un impianto di irrigazione che collega delle piante a vari rubinetti che possono erogare acqua. Tutti i rubinetti sono inizialmente chiusi, e bisogna capire quale rubinetto aprire per far arrivare più velocemente possibile l'acqua ad una data pianta che necessita di essere annaffiata. L'impianto è rappresentato tramite un grafo non orientato pesato $G = (V, E, w)$ in cui i vertici in V rappresentano rubinetti o piante, un arco $(u, v) \in E$ rappresenta un tubo di collegamento dal vertice u al vertice v, ed il peso $w(u, v)$ indica il tempo che l'acqua impiega per attraversare il tubo (u, v) (sotto l'assunzione che l'acqua impiega il medesimo tempo ad attraversare il tubo partendo dal vertice u o partendo dal vertice v). Progettare un algoritmo che dato il grafo non orientato pesato $G = (V, E, w)$, l'insieme $R \subseteq V$ dei rubinetti, e la pianta $p \in V$ da annaffiare, restituisce il rubinetto $r \in R$ da aprire per far arrivare il più velocemente possibile l'acqua alla pianta p.

4. **Esercizio.** Progettare un algoritmo che, dato un grafo orientato pesato $G = (V, E, w)$, e due sottoinsiemi disgiunti $V_1, V_2 \subseteq V$ (disgiunti vuol dire che $V_1 \cap V_2 = \emptyset$), restituisce il cammino minimo da V_1 a V_2 ovvero restituisce il cammino di costo minimo fra tutti i cammini che partono da un qualsiasi vertice in V_1 e terminano in un qualsiasi vertice in V_2.

5. **Esercizio.** Dato un grafo non orientato connesso $G = (V, E)$ ed un suo vertice $v \in V$, definiamo **raggio** del vertice v in G, la massima distanza fra v ed un qualsiasi altro vertice in V (si ricorda che la distanza fra due vertici è il numero minimo di archi di un cammino fra tali vertici). Matematicamente:

   $$raggio(v, G) = max\{u \in V \mid distanza(v, u)\}$$

   Progettare un algoritmo che dato un grafo non orientato connesso $G = (V, E)$ ed un vertice $v \in V$, restituisce $raggio(v, G)$.

6. **Esercizio.** Progettare un algoritmo che dato un grafo orientato pesato $G = (V, E, w)$ verifica se G contiene un ciclo di costo complessivo negativo.

7. **Esercizio.** Una comunità di formiche deve costruire più velocemente possibile un formicaio che permetta alle formiche di spostarsi fra un certo insieme di stanze S. Ogni stanza $s \in S$ richiede un tempo di costruzione $K(s)$. Per ogni coppia di stanze $s_1, s_2 \in S$, esiste un tempo $T(s_1, s_2)$ per costruire una galleria di collegamento diretto tra le due stanze (si ha che $T(s_1, s_2) = T(s_2, s_1)$). Il tempo complessivo di costruzione del formicaio è dato dalla somma dei tempi di costruzione delle gallerie più la somma dei tempi di costruzione delle stanze. Il formicaio non deve obbligatoriamente avere una galleria diretta fra ogni coppia di stanze, ma è sufficiente che per ogni coppia di stanze esista un cammino di collegamento, che potrebbe includere più gallerie e stanze intermedie. Progettare un algoritmo che dato l'insieme di stanze S, la funzione $K : S \to \mathbb{R}$ (tale che $\forall s \in S \,.\, k(s) \geq 0$), e la funzione $T : S \times S \to \mathbb{R}$ (tale che $\forall s_1, s_2 \in S \,.\, T(s_1, s_2) \geq 0$), restituisce il tempo minimo di costruzione del formicaio.

8. **Esercizio.** Una rete autostradale viene rappresentata tramite un grafo non orientato pesato in cui i vertici rappresentano città, gli archi rappresentano tratti autostradali di collegamento fra città, ed i pesi degli archi rappresentano i costi dei relativi pedaggi autostradali. Inoltre, ogni volta che si attraversa una città, è necessario pagare una tassa di attraversamento (oltre ai costi dei pedaggi autostradali dei tratti usati per raggiungere e per uscire dalla città). Quindi, dato un cammino $v_0, v_1, \ldots v_k$, il costo complessivo di tale cammino è dato dalla somma di tutti i pedaggi per i tratti (v_i, v_{i+1}), con $i \in \{0, \ldots, k-1\}$ più le tasse di attraversamento delle città v_j, con $j \in \{1, \ldots, k-1\}$. Progettare un algoritmo che dato un grafo non orientato pesato $G = (V, E, w)$ con pesi non negativi, una funzione di costo $t : V \to \mathbb{R}$ tale che $\forall v \in V \,.\, t(v) \geq 0$, un nodo sorgente $s \in V$ ed uno destinazione $d \in V$, stampa un cammino di costo complessivo minimo per andare da s a d.

Cont.

9. **Esercizio.** Dato un grafo orientato aciclico $G = (V, E)$, stampare tutti i vertici appartenenti a V in modo tale che se esiste un arco $(v_i, v_j) \in E$ allora al momento della stampa di v_i, il vertice v_j deve essere già stato stampato.

10. **Esercizio.** Progettare un algoritmo che, dato un grafo orientato $G = (V, E)$, verifica se tale grafo contiene almeno un ciclo.

11. **Esercizio.** Dato un grafo orientato aciclico $G = (V, E)$, stampare tutti i vertici appartenenti a V in modo tale che se dal vertice v_i è possibile raggiungere il vertice v_j, allora v_i viene stampato prima di v_j.

12. **Esercizio.** Progettare un algoritmo che, dato un grafo non orientato connesso $G = (V, E)$ e due vertici $v_1, v_2 \in V$, stampa un cammino di lunghezza minima da v_1 a v_2.

13. **Esercizio.** Dato un grafo orientato pesato $G = (V, E, w)$ e un vertice $v \in V$, stampare un ciclo di costo minimo che contiene v. In caso di assenza di un tale ciclo stampare la stringa `"no ciclo minimo"`.

14. **Esercizio.** Si consideri il seguente gioco per bambini che si sviluppa in un parco, su un campo di gioco che rappresentiamo tramite un grafo non orientato pesato $G = (V, E, w)$. I vertici V identificano dei punti sul campo di gioco. In ogni punto si colloca un gruppo di bambini. Un arco $(v_1, v_2) \in E$ identifica la presenza in v_1 di un bambino che può spostarsi verso il punto v_2. Per spostarsi, però, il bambino richiede un numero di caramelle $w(v_1, v_2)$. La maestra dei bambini sceglie un punto iniziale s del campo di gioco, comunica ai bambini nel punto s un messaggio segreto, e gli consegna un sacchetto con K caramelle.

    All'inizio del gioco solo i bambini nel punto s conoscono il messaggio segreto. Lo scopo è che tutti i bambini alla fine del gioco conoscano tale messaggio. La comunicazione del messaggio avviene attraverso passaparola, ovvero, un bambino che viene a conoscenza del messaggio nel punto v_1 può andare a comunicarlo ai bambini nel corrispondente punto v_2. Ogni bambino che si sposta da v_1 a v_2 può prendere con sè alcune caramelle, inizialmente presenti nel sacchetto della maestra, passarle poi ai bambini nel punto v_2, tenendone per sè un numero pari a $w(v_1, v_2)$. Lo scopo del gioco è quindi raggiungibile solo se il numero iniziale K di caramelle è sufficiente per permettere ad ogni bambino, che si sposta per il passaparola, di tenere per sè le proprie caramelle.

    Scrivere un algoritmo che dato il grafo (V, E, w), il punto di partenza s, ed il numero iniziale di caramelle K, verifica se è possibile raggiungere lo scopo del gioco.

Cont.

5 Soluzioni

5.1 Esercizi su Analisi del Costo Computazionale

1. **Soluzione.** L'algoritmo MYSTERY1 invoca MYSTERY2. Iniziamo quindi a calcolare il tempo di calcolo $T'(m)$ di MYSTERY2. La funzione MYSTERY2 è ricorsiva con costo $T'(m)$ che soddisfa la seguente relazione di ricorrenza:
$$T'(m) = \begin{cases} d' & \text{se } n \leq 1 \\ 2T'(m/4) + c' & \text{altrimenti} \end{cases}$$
Applicando il master theorem otteniamo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 4} = \frac{1}{2}$ e $\beta = 0$. Avendo $\alpha > \beta$, applichiamo il primo caso del master theorem ottenendo $T'(m) = \Theta(m^\alpha) = \Theta(m^{\frac{1}{2}})$. Ora consideriamo il costo $T(n)$ della funzione ricorsiva MYSTERY1 che soddisfa la seguente relazione di ricorrenza:
$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ 2T(n/3) + \Theta\big((n/2)^{\frac{1}{2}}\big) & \text{altrimenti} \end{cases}$$
Ma $\Theta\big((n/2)^{\frac{1}{2}}\big) = \Theta\big((1/2)^{\frac{1}{2}} \times n^{\frac{1}{2}}\big) = \Theta\big(n^{\frac{1}{2}}\big)$. Possiamo quindi applicare il master theorem ottenendo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 3} = \frac{1}{\log 3}$ e $\beta = \frac{1}{2}$. Avendo $\alpha > \beta$, applichiamo il primo caso del master theorem ottenendo $T(n) = \Theta(n^\alpha) = \Theta(n^{\frac{1}{\log 3}})$.

2. **Soluzione.** L'algoritmo MYSTERY1 utilizza MYSTERY2. Iniziamo quindi l'analisi da tale secondo algoritmo, che risulta essere un algoritmo ricorsivo con costo $T'(m)$ caratterizzato dalla seguente relazione di ricorrenza:
$$T'(m) = \begin{cases} c_1 & \text{se } m = 1 \\ T'(m-1) + c_2 & \text{altrimenti} \end{cases}$$
con c_1 e c_2 costanti. Si noti che tale relazione non si basa su partizionamenti bilanciati e quindi non è possibile l'utilizzo del Master Theorem. Procediamo con il metodo dell'iterazione:
$$T'(m) = T'(m-1) + c_2 = T'(m-2) + c_2 + c_2 = \cdots = T'(1) + (m-1) \times c_2 = c_1 + (m-1) \times c_2 = \Theta(m)$$
Analizziamo ora MYSTERY1. Tutte le operazioni hanno costo costante ad esclusione della invocazione a MYSTERY2 e dell'invocazione ricorsiva a MYSTERY1. Il tempo di calcolo $T(n)$ soddisfa quindi la seguente relazione di ricorrenza:
$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ T(n/2) + \Theta(n/2) & \text{altrimenti} \end{cases}$$
Applichiamo il Master Theorem. Abbiamo $\alpha = \frac{\log 1}{\log 2} = 0$ e $\beta = 1$. Visto che $\alpha < \beta$, per il terzo caso del teorema abbiamo $T(n) = \Theta(n^\beta) = \Theta(n)$.

3. **Soluzione.** Indichiamo con $T(n)$ e $T'(m)$ i costi computazionali delle invocazioni di funzioni MYSTERY(n) e MYSTERY2(m), rispettivamente. Visto che le due funzioni si richiamano vicendevolmente, abbiamo che:
$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ c_2 \times n + T'(n-1) & \text{altrimenti} \end{cases} \qquad T'(m) = \begin{cases} c'_1 & \text{se } m \leq 1 \\ c'_2 \times m + T(m-1) & \text{altrimenti} \end{cases}$$
Come primo passaggio, nella prima relazione, sostituiamo $T'(n-1)$ con la relativa definizione, ottenendo:
$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ c_2 \times 2 + c'_1 & \text{se } n = 2 \\ c_2 \times n + c'_2 \times (n-1) + T(n-2) & \text{altrimenti} \end{cases}$$
Ora abbiamo la seguente relazione di ricorrenza:
$$T(n) = \begin{cases} c' & \text{se } n \leq 2 \\ c_2 \times n + c'_2 \times (n-1) + T(n-2) & \text{altrimenti} \end{cases}$$
che risolviamo tramite iterazione:
$$\begin{aligned} T(n) &= c_2 \times n + c'_2 \times (n-1) + T(n-2) \\ &= \Theta(n) + \Theta(n-1) + T(n-2) \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \Theta(n-3) + T(n-4) \\ &= \cdots = \textstyle\sum_{i=1}^{n} \Theta(i) = \Theta(\frac{n \times (n+1)}{2}) = \Theta(n^2) \end{aligned}$$

Cont.

4. **Soluzione.** Indichiamo con $T(n)$ e $T'(m)$ i costi computazionali delle due invocazioni di funzioni MYSTERY(n) e MYSTERY2(m), rispettivamente. Visto che MYSTERY(n), se $n > 1$, esegue operazioni di costo costante all'interno di due for annidati di esattamente n cicli (oltre ad una chiamata ricorsiva con parametro $n/2$ ed una chiamata MYSTERY2(n)), e che MYSTERY2(m) effettua due chiamate MYSTERY($m/2$), abbiamo che:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + T'(n) + c_2 \times n^2 & \text{altrimenti} \end{cases} \qquad T'(m) = 2 \times T(m/2)$$

Come primo passaggio sostituiamo, nella prima relazione, $T'(n)$ con la relativa definizione, ottenendo:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + 2 \times T(n/2) + c_2 \times n^2 & \text{altrimenti} \end{cases}$$

Ora abbiamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 3 \times T(n/2) + c_2 \times n^2 & \text{altrimenti} \end{cases}$$

che risolviamo tramite Master Theorem. Abbiamo $a = 3$, $b = 2$ e $\beta = 2$; quindi $\alpha = \frac{\log 3}{\log 2} = \log 3$, e per il terzo caso 3 del Master Theorem (in quanto $\log 3 < 2$) possiamo concludere $T(n) = \Theta(n^2)$.

5. **Soluzione.** Iniziamo dall'analisi del costo computazionale $T'(m)$ della funzione MYSTERY2. Si tratta di una funzione che soddisfa la seguente relazione di ricorrenza:

$$T'(m) = \begin{cases} d & \text{se } m = 1 \\ 2 \times T'(\frac{m}{3}) + c \times m & \text{altrimenti} \end{cases}$$

in quanto la funzione è ricorsiva con costo costante nel caso base e, nel caso ricorsivo, con due chiamate ricorsive in cui si riduce di 3 volte il parametro più un ciclo di costo lineare. Tale relazione di ricorrenza può essere risolta utilizzando il Master Theorem considerando i parametri $a = 2$, $b = 3$ e $\beta = 1$ che implicano $\alpha = \frac{\log 2}{\log 3} = \frac{1}{\log 3}$. Avendo $\alpha < \beta$ ci troviamo nel caso 3 del teorema che implica $T'(m) = \Theta(m)$.

Passiamo ora all'analisi del costo computazionale $T(n)$ della funzione MYSTERY che esegue per $\log n$ volte un ciclo (in quanto ad ogni loop si raddoppia l'indice di controllo del ciclo) che contiene due chiamate alla funzione MYSTERY2 con parametro $n/2$. Abbiamo quindi $T(n) = \log n \times 2 \times T'(n/2) = \log n \times 2 \times \Theta(n/2) = \Theta(\log n \times n)$, che solitamente scriviamo come $\Theta(n \log n)$.

6. **Soluzione.** Iniziamo dall'analisi del costo computazionale $T'(m)$ della funzione MYSTERY2. Si tratta di una funzione che soddisfa la seguente relazione di ricorrenza:

$$T'(m) = \begin{cases} d' & \text{se } m = 1 \\ 2 \times T'(\frac{m}{3}) + c' \times m & \text{altrimenti} \end{cases}$$

in quanto la funzione è ricorsiva con costo costante nel caso base e, nel caso ricorsivo, con due chiamate ricorsive in cui si riduce di 3 volte il parametro più un ciclo di costo lineare. Tale relazione di ricorrenza può essere risolta utilizzato il Master Theorem considerando i parametri $a = 2$, $b = 3$ e $\beta = 1$ che implicano $\alpha = \frac{\log 2}{\log 3} = \frac{1}{\log 3}$. Avendo $\alpha < \beta$ ci troviamo nel caso 3 del teorema che implica $T'(m) = \Theta(m)$.

Passiamo ora all'analisi del costo computazionale $T(n)$ della funzione MYSTERY. Si tratta di una funzione che soddisfa la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{se } m = 1 \\ 3 \times T(\frac{n}{4}) + c \times \frac{n}{2} \times T'(\frac{n}{2}) & \text{altrimenti} \end{cases}$$

in quanto la funzione è ricorsiva con costo costante nel caso base e, nel caso ricorsivo, con tre chiamate ricorsive in cui si riduce di 4 volte il parametro più un ciclo eseguito $n/2$ volte (in quanto ad ogni loop si incrementa di 2 l'indice di controllo del ciclo) che contiene una chiamate alla funzione MYSTERY2 con parametro $n/2$. Abbiamo che $T'(\frac{n}{2}) = \Theta(\frac{n}{2})$ e quindi $\frac{n}{2} \times T'(\frac{n}{2}) = \Theta(n^2)$. La relazione di ricorrenza può quindi essere riscritta come segue:

$$T(n) = \begin{cases} d & \text{se } m = 1 \\ 3 \times T(\frac{n}{4}) + \Theta(n^2) & \text{altrimenti} \end{cases}$$

Tale relazione di ricorrenza può essere risolta utilizzato il Master Theorem considerando i parametri $a = 3$, $b = 4$ e $\beta = 2$ che implicano $\alpha = \frac{\log 3}{\log 4}$. Avendo $\alpha < \beta$ ci troviamo nel caso 3 del teorema che implica $T(n) = \Theta(n^2)$.

Cont.

7. **Soluzione.** Indichiamo con $T(n)$ e $T'(m)$, rispettivamente, i costi computazionali in tempo delle funzioni MYSTERY(n) e MYSTERY2(m). Iniziamo analizzando il costo in tempo di MYSTERY2(m) che risulta essere $T'(m) = \Theta(m)$ in quanto tale funzione include operazioni a costo costante ed un **for** che esegue m cicli. Riguardo a MYSTERY(n), il suo costo computazionale soddisfa la seguente relazione:

$$T(n) = \begin{cases} c & \text{se } n \leq 1 \\ 2T(n/2) + d \times n \times T'(n) & \text{altrimenti} \end{cases}$$

in quanto se $n \leq 1$ termina eseguendo una operazione **return** di costo costante, altrimenti esegue due chiamate ricorsive con parametro dimezzato con l'aggiunta di un ciclo **for** che invoca n volte MYSTERY2(n). Sostituiamo $d \times n \times T'(n)$ con $d \times n \times \Theta(n) = \Theta(n^2)$ e otteniamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c & \text{se } n \leq 1 \\ 2T(n/2) + \Theta(n^2) & \text{altrimenti} \end{cases}$$

che risolviamo tramite Master Theorem. Abbiamo $a = 2$, $b = 2$ e $\beta = 2$; quindi $\alpha = \frac{\log 2}{\log 2} = 1$, e per il terzo caso 3 del Master Theorem (in quanto $1 < 2$) possiamo concludere $T(n) = \Theta(n^2)$.

8. **Soluzione.** Indichiamo con $T(n)$ e $T'(m)$, rispettivamente, i costi computazionali in tempo delle funzioni MYSTERY(n) e MYSTERY2(m). Iniziamo analizzando il costo in tempo di MYSTERY(n); il suo costo computazionale soddisfa la seguente relazione:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ c_2 \times n + T'(n) & \text{altrimenti} \end{cases}$$

In quanto se $n \leq 1$ esegue una operazione di costo costante (**return**) mentre, in caso contrario, effettua n cicli di un **for** (con corpo del ciclo di costo costante) ed una chiamata a MYSTERY2 con parametro n. Applicando un simile ragionamento possiamo concludere che il costo computazionale di MYSTERY2(m) soddisfa la seguente relazione:

$$T'(m) = \begin{cases} d_1 & \text{se } m \leq 1 \\ d_2 \times n + T(m-1) & \text{altrimenti} \end{cases}$$

È possibile effettuare una sostituzione di $T'(n)$ nella relazione per la funzione $T(n)$ ottenendo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ c_2 \times n + d_2 \times n + T(n-1) & \text{altrimenti} \end{cases}$$

Osserviamo innanzitutto che $c_2 \times n + d_2 \times n = (c_2 + d_2) \times n = \Theta(n)$; possiamo quindi riscrivere la relazione di ricorrenza come segue:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ \Theta(n) + T(n-1) & \text{altrimenti} \end{cases}$$

Procediamo a risolvere tale relazione utilizzando il metodo dell'iterazione:

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) \\ &= \Theta(n) + \Theta(n-1) + T(n-2) \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + T(n-3) \\ &= \dots \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \dots + \Theta(2) + c_1 \end{aligned}$$

Avendo $c_1 = \Theta(1)$ possiamo concludere:

$$T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(\frac{n(n+1)}{2}) = \Theta(n^2)$$

Cont.

9. **Soluzione** L'algoritmo MYSTERY1 utilizza MYSTERY2. Iniziamo quindi l'analisi da tale secondo algoritmo, che risulta essere un algoritmo ricorsivo con costo $T'(m)$ caratterizzato dalla seguente relazione di ricorrenza:

$$T'(m) = \begin{cases} c_1 & \text{se } m = 1 \\ 2 \times T'(m/2) + c_2 & \text{altrimenti} \end{cases}$$

con c_1 e c_2 costanti. Infatti, la funzione MYSTERY2 contiene operazioni a costo costante escluse due chiamate ricorsive, ognuna con parametro uguale ad $m/2$. Applichiamo il Master Theorem. Abbiamo $\alpha = \frac{\log 2}{\log 2} = 1$ e $\beta = 0$. Visto che $\alpha > \beta$, per il primo caso del teorema, abbiamo $T'(m) = \Theta(m^\alpha) = \Theta(m)$.

Analizziamo ora MYSTERY1. Tutte le operazioni hanno costo costante ad esclusione della invocazione a MYSTERY2 e dell'invocazione ricorsiva a MYSTERY1. Il tempo di calcolo $T(n)$ soddisfa quindi la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T'(n) + T(n-1) + c_2 & \text{altrimenti} \end{cases}$$

Si noti che tale relazione non si basa su partizionamenti bilanciati e quindi non è possibile l'utilizzo del Master Theorem. Procediamo con il metodo dell'iterazione osservando che nel caso ricorsivo la relazione di ricorrenza indica che $T(n) = T'(n) + T(n-1) + c_2 = \Theta(n) + T(n-1) + c_2$:

$$T(n) = \Theta(n) + T(n-1) + c_2 = \Theta(n) + \big(\Theta(n-1) + T(n-2) + c_2\big) + c_2 = \cdots = \\ \Theta(n) + \Theta(n-1) + \cdots + \Theta(2) + c_1 + (n-1) \times c_2 = \textstyle\sum_{i=2}^{n} \Theta(i) + \Theta(1) + \Theta(n)$$

Considerando che $\sum_{i=2}^{n} \Theta(i) + \Theta(1) = \Theta(n^2)$, in quanto $\sum_{i=1}^{n} i = \frac{(n+1)\times n}{2}$, avremo che l'ultima espressione coincide con $\Theta(n^2) + \Theta(n) = \Theta(n^2)$. Concludendo, abbiamo che $T(n) = \Theta(n^2)$.

10. **Soluzione.** L'algoritmo MYSTERY esegue operazioni a costo costante a meno delle operazioni *insert* e $deleteMax$ su binary heap, che hanno entrambe un costo logaritmico rispetto alla dimensione dell'heap. Il corpo del **while** viene eseguito $\lfloor \log n \rfloor + 1$ volte in quanto l'indice i inizialmente ha valore 1 e viene raddoppiato ad ogni ciclo fino a che non supera n. Ad ogni ciclo viene aggiunto un elemento nell'heap. Essendo il costo di ogni *insert* logaritmico rispetto alla dimensione dell'heap (costante quando la dimensione è uguale a 0), e considerando che tale dimensione ad ogni ciclo è 0, 1, 2, ..., $\lfloor \log n \rfloor$, il costo complessivo del **while** risulta essere $O(1)+O(\log 1)+O(\log 2)+\cdots+O(\log\lfloor \log n \rfloor) = O(\log 1+\log 2+\cdots+\log\lfloor \log n \rfloor)$. È possibile trasformare la sommatoria di logaritmi in logaritmo di prodotti ottenendo $O(\log(1\times 2\times 3\times\cdots\times\lfloor \log n \rfloor)) = O(\log(\lfloor \log n \rfloor!))$. Il ciclo **for** si comporta in modo simmetrico, eseguendo $\lfloor \log n \rfloor$ cicli, all'interno dei quali si esegue l'operazione $deleteMax$, di costo logaritmica, sul binary heap che riduce ad ogni ciclo la propria dimensione, iniziando da $\lfloor \log n \rfloor$ e arrivando a 1. Ha quindi il medesimo costo computazionale $O(\log(\lfloor \log n \rfloor!))$. Complessivamente, il costo risulta quindi essere $T(n) = O(\log(\lfloor \log n \rfloor!))$.

11. **Soluzione.** L'operazione $heapify$ trasforma un array in un heap con costo lineare, quindi con costo computazionale $\Theta(n)$. Il ciclo **while** viene eseguito una quantità logaritmica di volte, in quanto ad ogni ciclo l'indice di controllo i viene raddoppiato. Le operazioni eseguite all'interno del ciclo sono a costo costante ($findMax$ e aggiornamento dell'indice i) oppure logaritmiche nella dimensione dell'heap ($deleteMax$). La dimensione dell'heap è superiormente limitato da n, quindi possiamo quantificare il costo di ogni $deleteMax$ con $O(\log n)$. Il costo complessivo del ciclo **while** è quindi $O(\log n \times \log n)$. Complessivamente abbiamo quindi il seguente costo computazionale per la funzione MYSTERY: $T(n) = \Theta(n) + O(\log n \times \log n)$. Abbiamo però che $\log n \times \log n = O(n)$; questo può essere compreso ricordando che $\log n = O(n^{\frac{1}{2}})$ e quindi anche $(\log n)^2 = O((n^{\frac{1}{2}})^2)$, da cui appunto segue $\log n \times \log n = O(n)$. Questo permette di concludere che $T(n) = \Theta(n)$.

12. **Soluzione.** Innanzitutto, si deve tenere in considerazione il fatto che secondo l'implementazione quickFind, le operazioni sulle strutture UnionFind hanno i seguenti costi computazionali: $makeSet$ e $find$ hanno costo costante $O(1)$, mentre *union* ha costo $O(n)$ nel caso pessimo (con n dimensione della struttura). Veniamo ora all'analisi del costo $T(n)$ dell'algoritmo. L'algoritmo esegue operazioni di costo costante ad esclusione dell'operazione *union*. Il primo ciclo esegue una quantità pari ad n di operazioni a costo costante, quindi con costo risultante $O(n)$, e costruisce una struttura UnionFind di dimensione n. Il secondo ciclo viene eseguito una quantità di volte pari a $\lceil \log n \rceil$. Il corpo del ciclo, nel caso pessimo, ha costo $O(n)$ visto che oltre ad operazioni di costo costante include l'operazione *union* eseguita sulla struttura UnionFind di dimensione n costruita dal primo ciclo. Complessivamente, tale secondo ciclo ha quindi costo $O(n \log n)$. Avremo quindi $T(n) = O(n) + O(n \log n) = O(n \log n)$.

Cont.

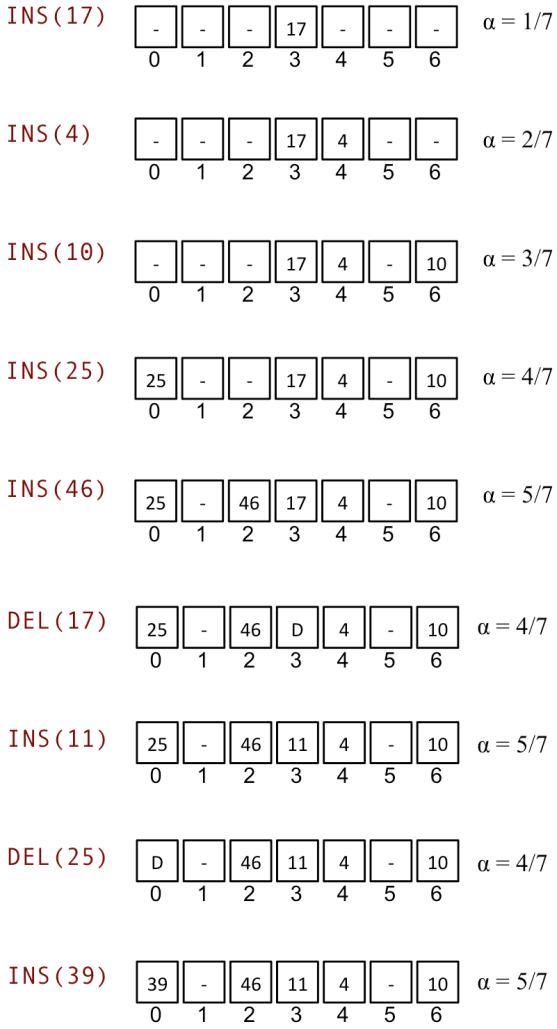
13. **Soluzione.** Nel caso di implementazione tramite quickUnion con euristica "by rank", le operazioni di *makeSet* e *union* hanno costo costante, mentre le operazioni di $find$ hanno costo logaritmico rispetto alla dimensione dell'insieme su cui viene effettuata l'operazione stessa. Il ciclo **for** iniziale esegue n operazioni *makeSet*, contribuisce quindi al costo computazionale per una quantità $\Theta(n)$. Le operazioni di assegnamento hanno costo costante. Il **while** viene eseguito $\Theta(logn)$ volte in quanto l'indice u viene raddoppiato ad ogni ciclo. Il costo del corpo del ciclo risulta essere $O(\log n)$ in quanto sono presenti operazioni a costo costante, più due esecuzioni di $find$ su insiemi che sono sicuramente superiormente limitati da n. Complessivamente il **while** contribuisce al costo computazionale con una quantità $O(\log n \times \log n)$. Infine, l'operazione **return** esegue una $find$ con costo $O(\log n)$. Sommando tutti questi contributi avremo che il costo computazionale della funzione MYSTERY risulta essere $T(n) = \Theta(n) + O(1) + O(\log n \times \log n) + O(\log n) = \Theta(n)$. Quest'ultimo passaggio è giustificato dal fatto che $\log n \times \log n = O(n)$; questo può essere compreso ricordando che $\log n = O(n^{\frac{1}{2}})$ e quindi anche $(\log n)^2 = O((n^{\frac{1}{2}})^2)$, da cui appunto segue $\log n \times \log n = O(n)$.

14. **Soluzione.** Nel caso di implementazione tramite QuickFind le operazioni di *makeSet* e $find$ hanno costo costante, mentre le operazioni di *union* hanno costo $O(n)$, con n dimensione della struttura UnionFind. Il ciclo **for** iniziale esegue n operazioni *makeSet*, contribuisce quindi al costo computazionale per una quantità $\Theta(n)$. Le operazioni di assegnamento hanno costo costante. Il **while** viene eseguito $\Theta(\log_4 n)$ volte in quanto l'indice u viene quadruplicato ad ogni ciclo. Il costo complessivo del **while** risulta essere $O(n \log n)$ in quanto all'interno del corpo del ciclo sono presenti operazioni a costo costante, più una esecuzione di *union* sulla struttura UnionFind che ha dimensione n (quindi con costo $O(n)$). Infine, l'operazione **return** esegue una $find$ con costo costante. Sommando tutti questi contributi avremo che il costo computazionale della funzione MYSTERY risulta essere $T(n) = \Theta(n) + O(1) + O(n \log n) + O(1) = O(n \log n)$.

Cont.

5.2 Esercizi su Strutture di Dati

1. **Soluzione.** La seguente immagine mostra la tabella hash e il fattore di carico (n/m) dopo ogni operazione:



Cont.

2. **Soluzione.** Per risolvere l'esercizio si può usare una visita ricorsiva "in parallelo" su entrambi gli alberi. Partendo dalle radici di T1 e T2, si verifica se i due nodi contengono lo stesso valore o sono entrambi vuoti. In caso contrario — quindi se uno solo dei due nodi è vuoto o i valori sono diversi — la visita si puó interrompere e il risultato sarà *false*. Se i due nodi hanno valori uguali, gli alberi saranno identici se (i) l'albero radicato nel figlio sinistro di T1 è identico all'albero radicato nel figlio sinistro di T2 e (ii) l'albero radicato nel figlio destro di T1 è identico all'albero radicato nel figlio destro di T2.

Algorithm 15: IDENTICI(NODO $T1$, NODO $T2$) $\rightarrow$ BOOLEAN

```
if T1 = null && T2 = null then
    return true
else if T1 = null || T2 = null then
    return false
else
    return (T1.val = T2.val) && IDENTICI(T1.left, T2.left) && IDENTICI(T1.right, T2.right)
```

Il costo computazione nel caso pessimo è $\Theta(n)$ dove n è il numero di nodi. Questo caso si verifica quando i due alberi sono identici, per cui l'algoritmo esegue (una sola volta) la visita in profondità di entrambi. Il caso ottimo è $O(1)$. Se si assume che l'operatore AND (&&) restituisca *false* appena una condizione è falsa (procedendo da sinistra verso destra) si verifica quando T1 è vuoto. Senza questa assunzione l'algoritmo richiede invece di visitare completamente entrambi gli alberi.

Cont.

3. **Soluzione.** L'esercizio si risolve con una visita in ampiezza. Nella coda usata per visitare i nodi si memorizza, oltre al valore del nodo, anche il livello in cui si trova. Ogni volta che si estrae un nodo si verifica se si è raggiunto un nuovo livello. In questo caso si verifica se i nodi del livello precedente erano in numero pari e il conteggio riparte per il nuovo livello. In caso contrario si aumenta il contatore dei nodi al livello corrente. Necessario un controllo finale per verificare se l'ultimo livello ha un numero di nodi pari.

   Chiamiamo tale algoritmo ContaLivelliPari. Visto che tale algoritmo richiede la visita dell'intero albero, e che per ogni nodo si eseguono operazioni di costo costante, il costo computazionale risulta essere $\Theta(n)$, con n dimensione dell'albero.

Algorithm 16: ContaLivelliPari(Nodo T) $\rightarrow$ Int

```
Int livelliConNodiPari ← 0
Queue q ← new Queue()
q.enqueue([T, 0])
Int livelloCorrente ← 0
Int nodiLivelloCorrente ← 0
while q.first ≠ null do
    /* estrae dalla coda una nuova coppia [N, l]                    */
    [N, l] ← q.dequeue()
    if l ≠ livelloCorrente then
        /* N è il primo nodo di un nuovo livello l                  */
        if nodiLivelloCorrente % 2 = 0 then
            livelliConNodiPari ← livelliConNodiPari + 1
        nodiLivelloCorrente ← 1
        livelloCorrente ← l
    else
        /* N è un ulteriore nodo dell'attuale livello corrente      */
        nodiLivelloCorrente ← nodiLivelloCorrente + 1
    for x ∈ N.children do
        q.enqueue([x, l + 1])
/* controllo relativo all'ultimo livello                            */
if nodiLivelloCorrente % 2 = 0 then
    livelliConNodiPari ← livelliConNodiPari + 1
return livelliConNodiPari
```

Cont.

4. **Soluzione** Per risolvere l'esercizio si può usare l'algoritmo NodiK che effettua una visita in profondità partendo dalla radice e passando come parametro l'altezza k. Ad ogni chiamata ricorsiva si decrementa il valore di k fino al valore 0; in questo caso il nodo raggiunto sarà a livello k per cui dovrà essere conteggiato e la visita può essere interrotta.

Algorithm 17: NodiK(Node T, Int k) → Int

```
if T = null then
    return 0
else
    if k = 0 then
        return 1
    else
        return NodiK(T.left, k-1) + NodiK(T.right, k-1)
```

Il caso ottimo si verifica se l'albero degenera in una sorta di lista (quindi se l'unico nodo ad ogni livello ha un solo figlio, o destro, o sinistro). In questo caso il costo computazionale è $\Theta(k)$ visto che è sufficiente attraversare k nodi. Il caso pessimo si verifica se l'albero è completo fino al livello k per cui è necessario attraversare tutti i nodi fino a quel livello. Visto che un albero binario completo di livello k ha $2^{k+1}-1$ nodi, il costo computazionale risulta essere $\Theta(2^k)$.

5. **Soluzione.** Implementiamo una funzione ricorsiva (ricerca in profondità) che decrementa il parametro k ad ogni chiamata ricorsiva. Il paramento k viene decrementato solo se è maggiore di zero (ricordiamo che la funzione deve prendere in input un intero positivo). Abbiamo due casi base:
   1) Il primo gestisce il caso in cui l'albero in input sia vuoto. In questo caso la funzione ritorna 0.
   2) Il secondo gestisce il caso in cui il nodo corrente sia una foglia e il parametro $k = 0$ (questo implica che abbiamo raggiunto/superato la profondità richiesta nella prima chiamata a funzione). In questo caso la funzione ritorna 1.

Algorithm 18: CountLeaves(BinTree T, Int k) → Int

```
if T = NULL then
    return 0
else if T.isLeaf() and k = 0 then
    return 1
else
    return CountLeaves(T.left,Max(0,k-1))+CountLeaves(T.right,Max(0,k-1))
```

Sia nel caso pessimo che nel caso ottimo siamo costretti a visitare tutti i nodi dell'albero per poter individuare le foglie a profondità maggiore o uguale a k. Il costo computazionale dell'algoritmo à quindi $\Theta(n)$, dove n è il numero di nodi nell'albero T.

Cont.

6. **Soluzione.** Una possibile soluzione, non particolarmente efficiente, effettua una scansione delle lista L_1 per ogni intero presente in L_2 (vedi Algoritmo 19). La funzione ListComplement1 esegue una chiamata alla funzione Remove per ogni elemento in L_2. Nel caso pessimo, la funzione di supporto Remove ha un costo pari al numero di interi nella lista L, O($|L|$), mentre nel caso ottimo (quando x è minore di ogni intero in L) ha un costo costante, O(1). Quindi, nel caso pessimo ListComplement1 costa O($|L_1||L_2|$) e nel caso ottimo O($|L_2|$).

Algorithm 19: ListComplement1(List L_1, List L_2) → List

```
while L2 ≠ NULL do
    L1 ← Remove(L1,L2.val)
    L2 ← L2.next
return L1

function Remove(List L, Int x) → List
if L = NULL or x < L.val then
    return L
else if L.val = x then
    // Necessario deallocare L qui se il linguaggio non dispone di un garbage collector
    return Remove(L.next,x)
else
    L.next ← Remove(L.next,x)
    return L
```

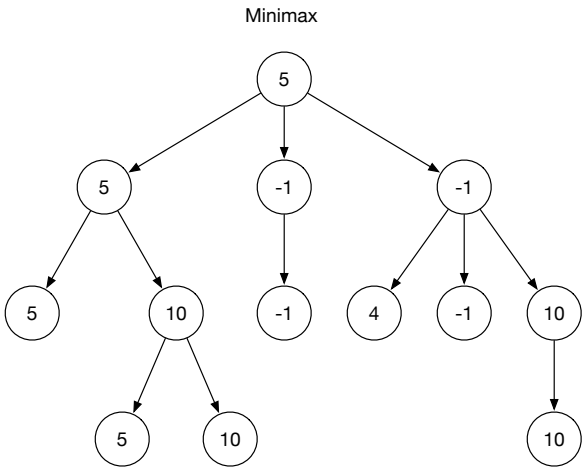
La funzione ListComplement1 non sfrutta interamente l'ordinamento imposto a L_1 ed L_2. Possiamo implementare una funzione maggiormente efficiente, che evita di visitare L_1 dall'inizio per ogni valore in L_2: se il nodo correntemente visitato in L_1 ha un valore maggiore di quello del nodo correntemente visitato in L_2, avanziamo su L_2; diversamente, se il nodo corrente in L_1 ha un valore minore, avanziamo su L_1; se i due valori sono uguali, rimuoviamo il nodo corrente in L_1 e proseguiamo a valutare il nodo successivo in L_1 senza avanzare su L_2. Per implementare questa seconda soluzione preferiamo una versione interamente ricorsiva ListComplement2 (vedi Algoritmo 20). Nel caso pessimo ListComplement2 visita tutti gli elementi di entrambe le liste (ad ogni step avanziamo o su L_1 oppure su L_2, mai su entrambe le liste contemporaneamente): costo O($|L_1| + |L_2|$). Nel caso ottimo tutti gli interi in L_1 sono minori degli interi in L_2 (o viceversa). In questo caso, la funzione scorre solo la lista con gli elementi minori: costo O($\min(|L_1|, |L_2|)$).

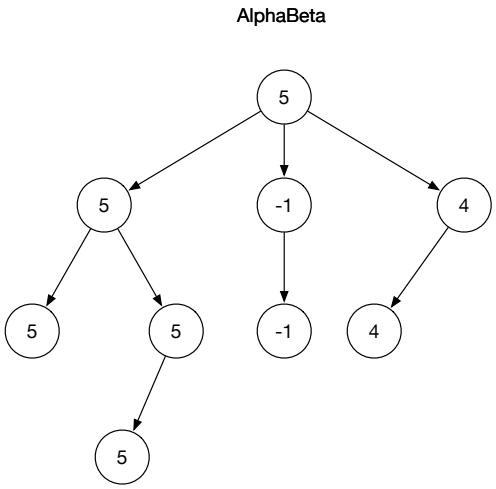
Algorithm 20: ListComplement2(List L_1, List L_2) → List

```
if L1 = NULL or L2 = NULL then
    return L1
else if L1.val = L2.val then
    // Necessario deallocare L1 qui se il linguaggio non dispone di un garbage collector
    return ListComplement2(L1.next,L2)
else if L1.val > L2.val then
    return ListComplement2(L1,L2.next)
else
    L1.next ← ListComplement2(L1.next,L2)
    return L1
```

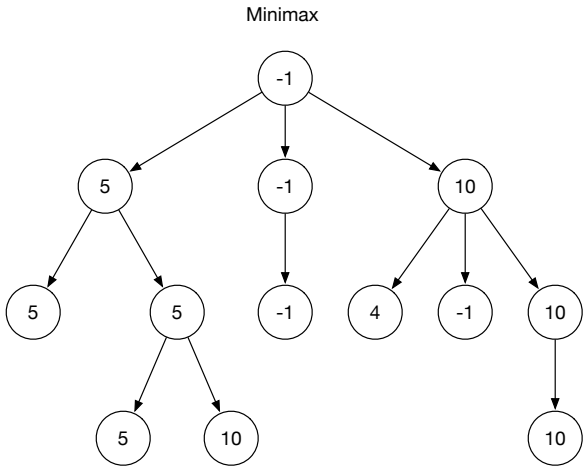
Cont.

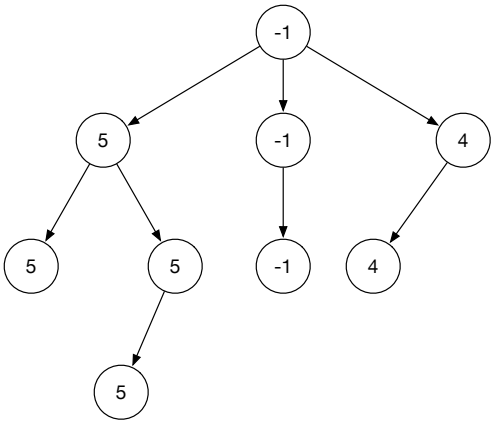
7. **Soluzione.** Gli alberi risultanti dopo l'applicazione dell'algoritmo Minimax e AlphaBeta pruning sono mostrati di seguito.





8. **Soluzione.** Gli alberi risultanti dopo l'applicazione dell'algoritmo Minimax e AlphaBeta pruning sono mostrati di seguito.





Cont.

9. **Soluzione.** Possiamo fornire diverse soluzioni al problema a seconda delle assunzioni che facciamo rispetto alla nostra struttura dati. Vediamo qui tre possibili approcci. Se assumiamo che i nodi del nostro albero binario di ricerca mantengano come informazione il numero di nodi nel proprio sottoalbero sinistro, una implementazione particolarmente efficiente è la seguente (Algoritmo 21).

Algorithm 21: kthSmallestKey1(BST T, Int k) → Int

```
if T = NULL then
 | return NA
else if k = T.leftNodes +1 then
 | return T.key
else if k < T.leftNodes then
 | return kthSmallestKey1(T.left,k)
else
 | return kthSmallestKey1(T.right,k-T.leftNodes-1)
```

Per individuare il k-esimo intero più piccolo, l'Algoritmo 21 effettua una visita su un percorso radice-foglia. Nel caso peggiore tale visita avrà una lunghezza pari all'altezza h dell'albero, quindi O(h).

Come seconda possibilità notiamo che in un albero binario di ricerca è sufficiente effettuare una in-visita e *contare* di volta in volta il numero di nodi visitati. Il k-esimo nodo visitato è il k-esimo nodo più piccolo nell'albero. Per poter implementare questo approccio è necessario assumere che il nostro linguaggio di programmazione ci permetta di passare argomenti per riferimento/indirizzo, in modo da poterne modificare il valore nelle chiamate ricorsive. Sotto tale assunzione, la seguente è una possibile soluzione al problema (Algoritmo 22). Nell'Algoritmo 22 facciamo uso della sintassi C per indicare il passaggio per indirizzo dell'argomento k.

Algorithm 22: kthSmallestKey2(BST T, Int *k) → Int

```
if T = NULL then
 | return NA
else
 | res ← kthSmallestKey2(T.left,k)
 | *k ← *k -1
 | if res ≠ NA then
 |  | return res
 | else if *k = 1 then
 |  | return T.key
 | else
 |  | return kthSmallestKey2(T.right,k)
```

L'Algoritmo 22 effettua una in-visita dell'albero binario di ricerca che si interrompe (ricorsione a destra non effettuata) non appena viene visitato il k-esimo nodo. Nel caso peggiore (quando k è maggiore del numero di nodi nell'albero) la visita procede per tutti ed n i nodi dell'albero ed ha quindi un costo pari a O(n).

Come terza soluzione consideriamo il caso in cui l'albero non contenga informazioni aggiuntive (come abbiamo assunto per l'Algoritmo 21) e che il linguaggio di programmazione ammetta solo funzioni con passaggio degli argomenti per valore (diversamente da quanto assunto per l'Algoritmo 22). In questo caso, una possibile soluzione efficiente è quella di cercare il valore minimo e poi individuare iterativamente il nodo successivo per $k-1$ volte (Algoritmo 23).

Cont.

Algorithm 23: kthSmallestKey3(BST T, Int k) → Int

```
node ← Min(T)
while k > 1 and node ≠ NULL do
    node ← Successor(node)
if node = NULL then
    return NA
else
    return node.key

function Min(BST T) → BST
if T = NULL or T.left = NULL then
    return T
else
    return Min(T.left)

function Successor(BST T) → BST
if T = NULL then
    return NULL
else if T.right ≠ NULL then
    return Min(T.right)
else
    P ← T.parent
    while P ≠ NULL and T = P.right do
        T ← P
        P ← P.parent
    return P
```

Individuare il minimo ed il successore di un nodo in un albero binario di ricerca costa nel caso pessimo $O(h)$, dove h è l'altezza dell'albero. Effettuiamo sempre una ricerca del minimo e $k-1$ ricerche del successore, quindi il costo computazionale nel caso pessimo dell'Algoritmo 23 è $O(kh)$.

Cont.

10. **Soluzione.** Dato un albero binario T, calcoliamo innanzitutto la profondità del nodo più a sinistra che non abbia un figlio sinistro (Algorithmo 24).

Algorithm 24: LeftDepth(Tree T) → Int

```
if T = NULL then
    return 0
else if T.isLeaf() then
    return 1
else
    return 1 + LeftDepth(T.left)
```

Notiamo che se l'albero T è perfetto la profondità del nodo più a sinistra coincide con la profondità di tutte le foglie di T. Conoscendo tale profondità possiamo facilmente verificare le condizioni 1 e 2 che caratterizzano un albero perfetto (Algoritmo 25).

Algorithm 25: Perfect(Tree T, Int *depth*) → Bool

```
if T.isLeaf() then
    return depth = 0
else if T.left = NULL or T.right = NULL then
    return FALSE
else
    return Perfect(T.left,depth-1) and Perfect(T.right,depth-1)
```

Mettendo insieme gli algoritmi 24 e 25 otteniamo la soluzione al nostro problema (Algoritmo 26). Assumiamo che un albero vuoto sia perfetto.

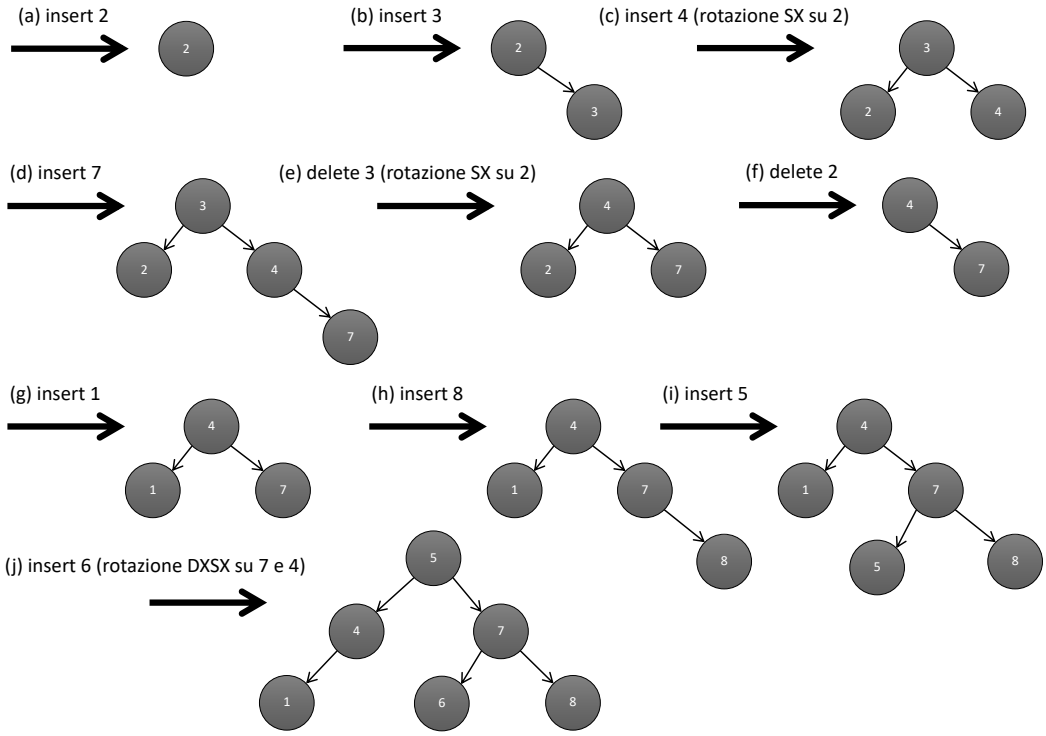
Algorithm 26: PerfectTree(Tree T) → Bool

```
if T = NULL then
    return TRUE
else
    return Perfect(T,LeftDepth(T))
```

La funzione LeftDepth (Algoritmo 24) nel caso pessimo attraversa il percorso più lungo radice-foglia, quindi ha un costo O(h), dove h è l'altezza dell'albero, mentre nel caso ottimo (quando la radice non ha figli sinistri) ha un costo costante O(1). La funzione Perfect (Algoritmo 25) nel caso pessimo deve visitare tutti i nodi dell'albero per raggiungere le foglie, quindi ha un costo O(n), dove n è il numero di nodi nell'albero, mentre nel caso ottimo termina immediatamente (costo costante). Poichè $h \leq n$, concludiamo che PerfectTree (Algoritmo 26) ha nel caso pessimo un costo pari a O(n) O(1) nel caso ottimo (ad esempio, albero la cui radice non abbia nessun figlio sinistro).

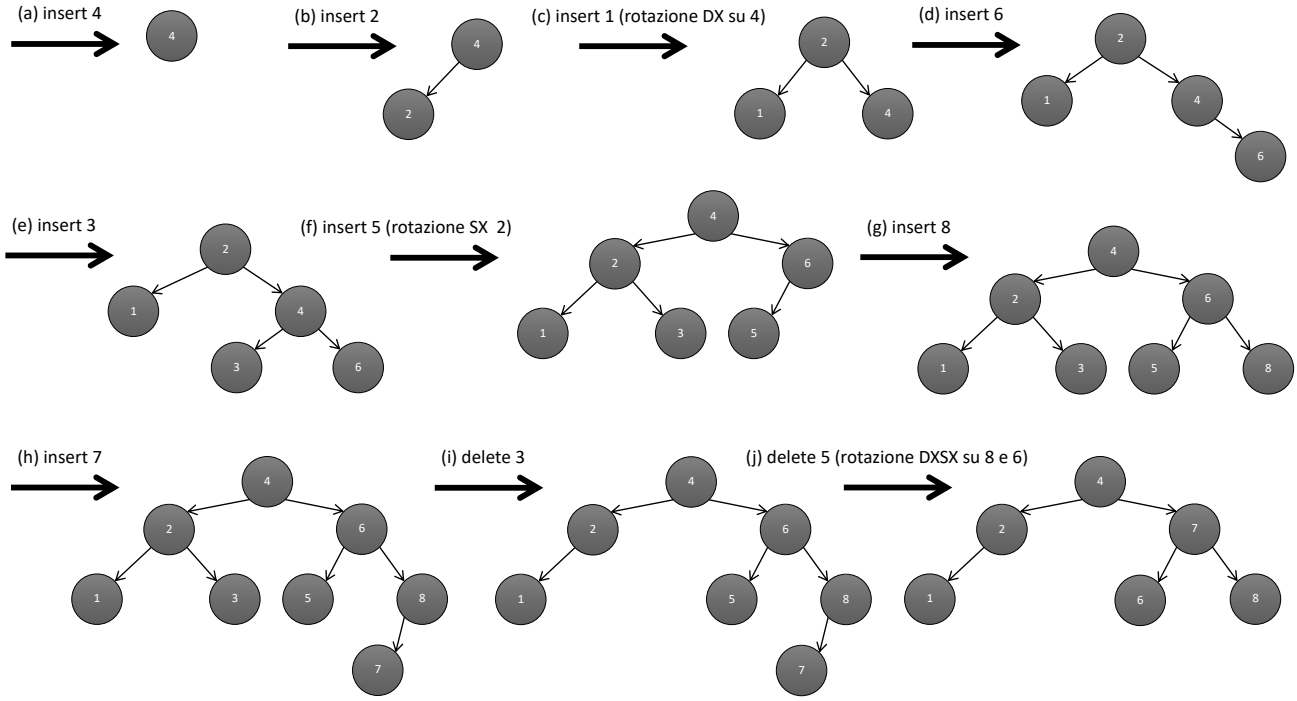
Cont.

11. **Soluzione.**



Cont.

12. **Soluzione.**



Cont.

13. **Soluzione.** Implementiamo una funzione iterativa che scorre la lista con due puntatori, uno *lento* ed uno *veloce*. Il puntatore lento scorre i nodi sequenzialmente mentre quello veloce procede saltando di due nodi per volta. Non appena il puntatore veloce ha finito di scorrere la lista, il puntatore lento sarà posizionato sul nodo medio nella lista, quello da rimuovere. Essendo una lista monodirezionale, per poter eliminare il nodo puntato dal puntatore lento è necessario mantenere traccia del nodo che lo precede (puntatore *tmp*). Inoltre, in modo da poter gestire il caso in cui la lista consista di un solo nodo (che verrà rimosso dalla procedura), facciamo ritornare alla procedura il puntatore alla testa della lista (se c'è un solo nodo la funzione ritornerà NULL).

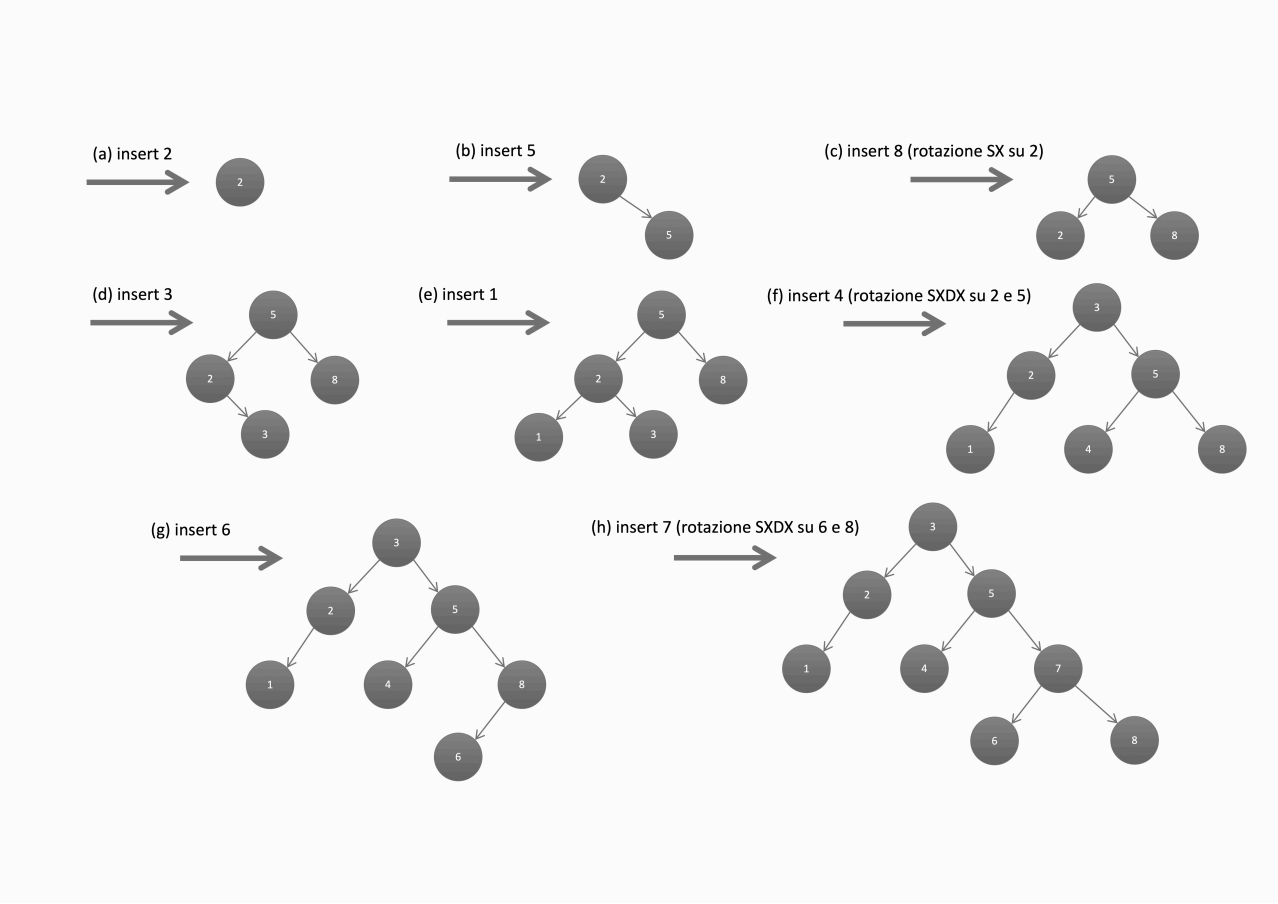
Algorithm 27: RemoveMidNode(List L) → List

```
if L = NULL or L.next = NULL then
    Free(L)
    return NULL
else
    List tmp ← NULL
    List slow ← L
    List fast ← L
    while fast ≠ NULL and fast.next ≠ NULL do
        tmp ← slow
        slow ← slow.next
        fast ← fast.next.next
    tmp.next ← slow.next
    Free(slow)
    return L
```

I casi *lista vuota* e *lista con un solo nodo* sono gestiti in modo istantaneo. Se la lista contiene almeno due nodi, la procedura la visita interamente una sola volta quindi il costo computazionale è $\Theta(n)$.

Cont.

14. **Soluzione.**



Cont.

5.3 Esercizi su Tecniche Algoritmiche

1. **Soluzione.** Il problema può essere risolto tramite un approccio greedy, abbinando l'i-esima femmina con l'i-esimo maschio. Per calcolare il tempo per completare la formazione delle coppie, si sommano i passi che le femmine devono fare per raggiungere il proprio maschio abbinato. Gli abbinamenti vengono calcolati inserendo gli indici delle posizioni delle ballerine, e dei ballerini, in due distinte code FIFO. Successivamente si estraggono gli indici abbinati e si calcola la distanza considerando il valore assoluto della differenza degli indici.

Algorithm 28: BALLETTO(BOOL $B[1..2n]$)→INT

```
QUEUE maschi ← new QUEUE()
QUEUE femmine ← new QUEUE()
for j ← 1 to 2n do
    if B[j] then
        femmine.enqueue(j)
    else
        maschi.enqueue(j)
tot ← 0
for j ← 1 to n do
    tot ← tot+ABS(femmine.dequeue() − maschi.dequeue())
return tot
```

Il costo computazionale di tale algoritmo risulta essere $T(n) = \Theta(n)$, assumendo che le operazioni di *queue* ed *enqueue* abbiano costo costante, visto che tutte le operazioni hanno tempo costante e vista la presenza di due **for** che eseguono entrambi $2n$ cicli.

Cont.

2. **Soluzione.** Al fine di massimizzare il peso medio delle caramelle, l'obiettivo è minimizzare il numero di caramelle usate per raggiungere il peso complessivo K. È possibile calcolare il numero minimo di caramelle per raggiungere il peso complessivo K tramite programmazione dinamica. Usiamo $P(i,j)$ per indicare il numero minimo di caramelle, tra quelle dei distributori con indice minore o uguale a i, per raggiungere il peso complessivo j. Usiamo $P(i,j) = \infty$ per indicare che non è possibile raggiungere il peso complessivo j. Tali problemi $P(i,j)$ possono essere risolti nel seguente modo:

$$P(i,j) = \begin{cases} j/p[1] & \text{se } i = 1 \text{ e } j\%p[1] = 0 \\ \infty & \text{se } i = 1 \text{ e } j\%p[1] \neq 0 \\ P(i-1,j) & \text{se } i > 1 \text{ e } p[i] > j \\ min\{P(i-1,j), 1 + P(i,j-p[i])\} & \textit{altrimenti} \end{cases}$$

Una volta risolto il problema $P(n,K)$, ovvero calcolato il numero minimo di caramelle che permettono di ottenere il peso complessivo K, il peso medio massimo sarà $K/P(n,K)$, che risulterà 0 nel caso in cui non sia possibile riempire il bicchiere esattamente per la sua capacità K.

Il problema può essere risolto dal seguente algoritmo che utilizza la struttura dati ausiliaria $B[i,j]$ per memorizzare le soluzioni ai problemi $P(i,j)$.

Algorithm 29: BICCHIERE(INT K,INT $p[1..n]$) $\rightarrow$ REAL

```
for j ← 1 to K do
    if j%p[1] = 0 then
        B[1, j] ← j/p[i]
    else
        B[1, j] ← ∞
for i ← 1 to n do
    for j ← 1 to K do
        if p[i] > j then
            B[i, j] ← B[i − 1, j]
        else
            B[i, j] ← min{B[i − 1, j], 1 + B[i, j − p[i]]}
return K / B[n,K]
```

Tutte le operazioni di tale algoritmo hanno costo costante; il primo ciclo **for** ha un contributo sul costo computazionale pari a $\Theta(K)$, mentre i due **for** annidati hanno un contributo pari a $\Theta(n \times K)$. Avremo quindi complessivamente un costo computazionale $T(K,n) = \Theta(n \times K)$.

Cont.

3. **Soluzione.** Una possibile soluzione prevede di ordinare il secondo vettore al fine di poter effettuare ricerche di elementi con costo logaritmico tramite ricerca binaria. Una volta ordinata sarà sufficiente scorrere gli elementi del primo vettore, e incrementare un contatore ogni volta che si incontra un elemento che è presente anche nel secondo vettore (usando appunto una ricerca binaria). L'algoritmo CONTA utilizza la variabile ausiliaria *conta* come contatore. Inoltre, l'algoritmo usa un algoritmo di ordinamento SORT che non specifichiamo; assumiamo che sia un algoritmo di ordinamento ottimale (ad esempio heapsort) di costo $O(n \log n)$.

   Studiamo ora il costo computazionale $T(n)$ dell'algoritmo CONTA iniziando l'analisi dall'algoritmo ausiliario di ricerca binaria RICERCA. Tale algoritmo ha un costo $O(\log n)$. L'algoritmo CONTA esegue prima l'ordinamento di costo $O(n \log n)$. Successivamente esegue un ciclo in cui invoca per n volte RICERCA, per un costo complessivo $O(n \log n)$. Complessivamente avremo $T(n) = 2 \times O(n \log n) = O(n \log n)$.

Algorithm 30: CONTA(INT $A[1..n]$, INT $B[1..n]$) $\rightarrow$ INT

```
SORT(B)
INT conta ← 0
for i ← 1 to n do
    if RICERCA(A[i], B, 1, n) then
        conta ← conta + 1
return conta

/* funzione di ricerca binaria in array ordinato                    */
function RICERCA(INT x, INT V[1..n], INT s, INT e) → BOOL
if s > e then
    return false
else
    INT m ← ⌊(s+e)/2⌋
    if x = V[m] then
        return true
    else
        if x < V[m] then
            return RICERCA(x, B, s, m − 1)
        else
            return RICERCA(x, B, m + 1, e)
```

Cont.

4. **Soluzione** Si può procedere secondo un criterio greedy, riempiendo i contenitori di capacità inferiore senza utilizzare (a meno che non sia necessario) quelli di capacità superiore. Questo permette di massimizzare il numero di contenitori utilizzati. Per sapere quali contenitori usare, si può inizialmente creare un min-heap contenente coppie di valori $(V[i], i)$, per $i \in \{1, \ldots, n\}$, con le capacità $V[i]$ usate come chiavi. Successivamente vengono estratti tali coppie dall'heap (quindi vengono estratte in ordine crescente di capacità) fino a raggiungere complessivamente una capacità sufficiente per contenere l'intera quantità K di gas.

   L'algoritmo RiempiContenitori utilizza una struttura intermedia A che viene riempita con le coppie $(V[i], i)$ e successivamente viene costruito un minHeap H, tramite heapify che considera come chiavi i primi valori delle coppie. Successivamente si estraggono dall'heap coppie (v, j), si riempiono i contenitori indicati dall'indice j, fino a svuotamento del gas iniziale. Si usa una variabile *res* per indicare il gas rimanente dopo ogni operazione di svuotamento.

Algorithm 31: RiempiContenitori(Int K, Real[1..n] V) $\rightarrow$ Real[$1 \ldots n$]

```
/* Inizializzazione:  res gas residuo, O vettore in output, A coppie (V[i], i)   */
Real res ← K
Real[1..n] O
(Real,Int)[1..n] A
for i ← 1 to n do
    A[i] ← (V[i], i)
    O[i] ← 0

/* Generazione dell'heap contenente le coppie (V[i], i) ordinate secondo il primo campo   */
minHeap[(Real,Int)]H ← A.minHeapify

/* Estrazione dall'heap dei contenitori fino a svuotamento completo del gas   */
while res > 0 do
    (Real,Int) ( v, j ) ← H.findMin()
    H.deleteMin()
    O[j] ← min(res, v)
    res ← res − v
return O
```

L'algoritmo esegue operazioni di costo costante ad esclusione della operazione di minHeapify e di deleteMin. La prima ha costo $O(n)$, mentre ogni deleteMin richiede tempo $O(\log n)$. Nel caso pessimo si dovranno eseguire n operazioni di deleteMin, per un costo complessivo $T(n) = O(n \log n)$.

Cont.

5. **Soluzione.** Si può utilizzare un approccio greedy che trova il massimo bottino semplicemente sommando i k oggetti di maggiore valore. Un modo efficiente, nel caso medio, per selezionare i k oggetti di maggiore valore utilizza una versione di quickselect per cercare il $(n-k)$-esimo minimo utilizzando sempre lo stesso array, partizionandolo utilizzando il cosiddetto algoritmo della bandiera nazionale discusso a lezione; al termine della ricerca dell'$(n-k)$-esimo minimo, tale algoritmo colloca nelle ultime k posizioni dell'array i k elementi di maggiore valore. Dopo aver eseguito questa versione di quickselect per cercare l'$(n-k)$-esimo minimo, sarà sufficiente sommare il valore degli elementi che si trovano nelle ultime k posizioni.

Algorithm 32: SommaMassimi(Int n, Int k, Number $v[1..n]$) → Number

```
s ← 1; e ← end     // s ed e delimitano il sottovettore su cui si sta effettuando la ricerca
while true do
    scegli casualmente un valore v[w] con w compreso tra s e e
    (iniziox, finex) ← Partiziona(v, s, e, v[w])
    if n − k < iniziox then
        e ← iniziox − 1                  // l'elemento scelto è maggiore dell'(n − k)-esimo minimo
    else if n − k > finex then
        s ← finex + 1                    // l'elemento scelto è minore dell'(n − k)-esimo minimo
    else
        exit                                          // l'elemento scelto è il (n − k)-esimo minimo
/* restuituice la sommatoria dei valori nelle ultime k posizioni di v                          */
somma ← 0
for i ← n − k + 1 to n do
    somma ← somma + v[i]
return somma
```

Algorithm Partiziona(Int $v[1..n]$, Int *start*, Int *end*, Number x) → (Int, Int)

```
/* bandiera nazionale:  minori di x verdi, uguali a x bianchi, maggiori di x rossi     */
/* restituisce una coppia di indici:  inizio e fine della collocazione dei valori x    */
i ← start; j ← start; k ← end
while j < k do
    if A[j] < x then
        if i < j then
            scambia A[i] con A[j]
        i ← i + 1; j ← j + 1
    else if A[j] > x then
        scambia A[j] con A[k]
        k ← k − 1
    else
        j ← j + 1                                                  // in questo caso A[j] = x
return (i, j − 1)          // i valori x (bianchi) sono dalla posizione i alla posizione j − 1
```

L'algoritmo SommaMassimi (si veda Algoritmo 32) è una versione iterativa di quickselect che ha quindi il medesimo costo computazionale studiato a lezione. Infatti il ciclo finale (di costo $\Theta(k)$) non cambia la classe di complessità, che risulta quindi essere lineare nel caso ottimo e medio, quadratica nel caso pessimo.

Cont.

6. **Soluzione.** È possibile procedere utilizzando programmazione dinamica, considerando i seguenti sottoproblemi: $P(i,j)$, con $i \in \{1,\ldots,n\}$ e $j \in \{0,\ldots,K\}$, che indica il numero massimo di pezzi di polistirolo delle prime i tipologie, che possono essere usati per ottenere esattamente un volume j. Nel caso in cui cui non sia possibile ottenere esattamente tale volume, poniamo $P(i,j) = -1$. Tali problemi possono essere risolti induttivamente rispetto a i nel seguente modo:

$$P(i,j) = \begin{cases} 0 & \text{se } i = 1 \text{ e } j = 0 \\ -1 & \text{se } i = 1,\ j > 0 \text{ e } j\%v[i] \neq 0 \\ j/v[i] & \text{se } i = 1,\ j > 0 \text{ e } j\%v[i] = 0 \\ P(i-1,j) & \text{se } i > 1 \text{ e } j < v[i] \\ max\{P(i-1,j), 1 + P(i,j-v[i])\} & \text{altrimenti} \end{cases}$$

Procediamo quindi a progettare un algoritmo (si veda Algoritmo 33) che risolve i problemi $P(i,j)$ memorizzando le relative soluzioni in una tabella $T[1..n, 0..K]$. Per poter poi ricostruire la combinazione di pezzi di polistirolo che genera la soluzione ottima, si utilizza una ulteriore tabella booleana $B[1..n, 0..K]$ tale che $B[i,j] = true$ se e solo se un pezzo di polistirolo di tipo i fa parte di una possibile soluzione ottima del problema $P(i,j)$. L'algoritmo proposto ha costo computazionale $T(K,n) = \Theta(n \times K)$.

Algorithm 33: POLISTIROLO(INT K,INT $v[1..n]$) $\rightarrow$ INT[1..N]

```
// Inizializzazione prima riga delle tabelle T e B
T[1,0] ← 0; B[1,0] ← false
for j ← 1 to K do
    if j%p[1] ≠ 0 then
        T[1,j] ← −1; B[1,j] ← false
    else
        T[1,j] ← j/p[i]; B[1,j] ← true
// Riempimento restanti righe delle tabelle T e B
for i ← 2 to n do
    for j ← 0 to K do
        if j < v[i] then
            T[i,j] ← T[i−1,j]; B[i,j] ← false
        else
            T[i,j] ← 1 + T[i,j−v[i]]; B[i,j] ← true
// Costruzione della soluzione ottimale x
for i ← 1 to n do
    x[i] ← 0
if T[n,K] ≠ −1 then
    // Almeno una soluzione ottima esiste
    i ← n; volumeRimanente ← K
    while volumeRimanente > 0 do
        if B[i, volumeRimanente] then
            x[i] ← x[i] + 1
            volumeRimanente ← volumeRimanente − v[i]
        else
            i ← i − 1
return x
```

Cont.

7. **Soluzione.** È possibile utilizzare la programmazione dinamica considerando i problemi $P(i)$, con $i \in \{1,\ldots,n\}$, tale che $P(i)$ = sequenza massima di indici $j_1, j_2, \ldots, j_k \in \{1 \ldots i\}$, tale che $j_l < j_{l+1}$ e $V[j_l] \geq V[j_{l+1}]$, per ogni $i \in \{1,\ldots,k-1\}$, ed inoltre $j_k = i$. Si noti quest'ultima condizione: nel problema $P(i)$ consideriamo sequenze che terminano in posizione i.

   Tali problemi $P(i)$ possono essere risolti considerando che:

$$P(i) = \begin{cases} 1 & \text{se } i = 1 \\ 1 + max\{P(j) \mid 1 \leq j < i \text{ and } V[j] \geq V[i]\} & \text{se } i > 1 \end{cases}$$

   Per stampare la sequenza di indici più lunga si utilizza un array ausiliario *prec*, tale che $prec[i]$ conterrà, per ogni $i \in \{1,\ldots,n\}$, il penultimo indice della sottosequenza ottimale per il problema $P(i)$. Gli indici della sequenza massima possono quindi essere reperiti a ritroso partendo dall'indice finale della sequenza più lunga.

Algorithm 34: MassimaSequenza(Int $V[1..n]$)

```
// Inizializzazione
Int prec[1..n], A[1..n], maxIndice, precIndice
A[1] ← 1; prec[1] ← −1; maxIndice ← 1      // maxIndice indicherà il j_k della sequenza massima
// Risoluzione sottoproblemi P(i)
for i ← 2 to n do
    // Ricerca del penultimo indice della sequenza del problema P(i)
    precIndice ← −1
    for j ← 1 to i − 1 do
        if V[j] ≥ V[i] and (precIndice = −1 or A[j] > A[precIndice]) then
            precIndice ← j
    // aggiornamento strutture dati
    if precIndice = −1 then
        A[i] ← 1
    else
        A[i] ← A[precIndice] + 1
    prec[i] ← precIndice
    if A[i] > A[maxIndice] then
        maxIndice ← i
// stampa della sequenza di lunghezza massima
printPath(prec, maxIndice)

// funzione ausiliaria di stampa del cammino
procedure printPath(Int prec[1..|V|], Int i)
if i ≠ −1 then
    printPath(prec, prec[d])
    print i
```

L'Algoritmo 34 risolve i problemi $P(i)$ inserendo le relative soluzioni nell'array $A[1..n]$, e poi procede per la stampa della sequenza di lunghezza massima trovata. Il costo computazionale $T(n)$ di tale algoritmo è dato dai due cicli **for** annidati in quanto le altre operazioni impiegano tempo costante e gli altri cicli sono eseguiti una quantità inferiore di volte. Il ciclo interno dei due **for** annidati viene eseguito un numero di volte pari a $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$ che implica $T(n) = \Theta(n^2)$.

Cont.

8. **Soluzione.** È possibile utilizzare la programmazione dinamica considerando i problemi $T(i,j)$, con $i \in \{1,\ldots,n\}$ e $j \in \{1,\ldots,W\}$, tale che $T(i,j)$ = valore complessivo minimo di un insieme di oggetti nell'insieme $\{1,\ldots,i\}$ con peso complessivo maggiore o uguale a j (∞ nel caso in cui non esista un tale insieme). I problemi $T(i,j)$ possono essere risolti considerando che:

$$T(i,j) = \begin{cases} \infty & \text{se } i = 1 \text{ e } P[1] < j \\ V[1] & \text{se } i = 1 \text{ e } P[1] \geq j \\ min\{V[i], T(i-1,j)\} & \text{se } i > 1 \text{ e } P[i] \geq j \\ min\{V[i] + T(i-1, j-P[i]), T(i-1,j)\} & \text{se } i > 1 \text{ e } P[i] < j \end{cases}$$

L'Algoritmo 35 risolve i problemi $T(i,j)$ inserendo le relative soluzioni nella tabella $T[1..n, 1..W]$. L'esercizio richiede di stampare l'insieme di oggetti ottimale (e non il loro valore memorizzato in $T[n,W]$); a tal fine l'algoritmo utilizza una matrice di booleani $B[i,j]$ che indica se l'oggetto i-esimo fa parte della soluzione al problema $T(i,j)$. Abbiamo che il costo computazionale di tale algoritmo risulta essere $T(n,W) = \Theta(n \times W)$ in quanto le operazioni elementari hanno tutte costo costante, ed il blocco di maggiore costo è quello che contiene i due **for** annidati le cui operazioni interne vengono eseguite $(n-1) \times W$ volte.

Algorithm 35: PALLONE(INT $P[1..n]$, INT $V[1..n]$, INT W)

```
// Inizializzazione prima riga delle tabelle T e B
for j ← 1 to W do
    if P[1] < j then
        T[1,j] ← ∞; B[1,j] ← false
    else
        T[1,j] ← V[1]; B[1,j] ← true
// Riempimento restanti righe delle tabelle T e B
for i ← 2 to n do
    for j ← 1 to W do
        if P[i] ≥ j and V[i] < T[i-1,j] then
            T[i,j] ← V[i]; B[i,j] ← true                    // si prende solo l'oggetto i-esimo
        else if P[i] < j and V[i] + T[i-1,j-P[i]] < T[i-1,j] then
            T[i,j] ← V[i] + T[i-1,j-P[i]]; B[i,j] ← true    // si prende anche l'oggetto i-esimo
        else
            T[i,j] ← T[i-1,j]; B[i,j] ← false               // non si prende l'oggetto i-esimo
if T[n,W] ≠ ∞ then
    // Esiste una soluzione e viene stampata
    pesoRimanente ← W; oggetto ← n
    while pesoRimanente > 0 do
        if B[oggetto, pesoRimanente] then
            print(oggetto)
            pesoRimanente ← pesoRimanente − P[oggetto]
        oggetto ← oggetto − 1
```

Cont.

9. **Soluzione.** È possibile procedere secondo un algoritmo greedy, semplicemente selezionando le trasmissioni in ordine crescente di *fine* dell'intervallo, facendo attenzione a selezionare trasmissioni che non si sovrappongono. Ad esempio, date le tre trasmissione dell'esempio riportato nel testo, $[4, 7]$, $[1, 5]$, $[2, 3]$, si riordinano nel modo seguente: $[2, 3]$, $[1, 5]$, $[4, 7]$. Poi si selezionano le trasmissioni nel seguente ordine: prima la trasmissione con intervallo $[2, 3]$, poi si scarta la trasmissione con intervallo $[1, 5]$ in quanto ha inizio precedente all'istante 3, e poi si seleziona la trasmissione con intervallo $[4, 7]$.

   L'Algoritmo 36 prende in input gli intervalli delle n trasmissioni rappresentate tramite un array contenente n coppie di numeri. Si utilizzano i metodi $first$ e $second$ per accedere al primo e al secondo elemento delle coppie. Il costo computazionale dell'algoritmo include il costo dell'ordinamento che, assumendo un algoritmo ottimale quale heapsort, risulta essere $O(n \log n)$, a cui si aggiunge il costo del ciclo che risulta essere $O(n)$. Complessivamente avremo quindi $T(n) = O(n \log n) + O(n) = O(n \log n)$.

Algorithm 36: TRASMISSIONI(NUMBERPAIRS $T[1..n]$)

```
T.sort(second)        // Ordina gli intervalli in modo crescente rispetto al secondo elemento
numTrasmissioni ← 0; fineUltimaTrasmissione ← 0; i ← 1
while i ≤ n do
    if T[i].first > fineUltimaTrasmissione then
        // Si seleziona la i-esima trasmissione
        numTrasmissioni ← numTrasmissioni + 1
        fineUltimaTrasmissione ← T[i].second
    i ← i + 1
return numTrasmissioni
```

Cont.

10. **Soluzione.** Il problema richiede di verificare se esiste un sottoinsieme di valori presi da un array di interi $A[1..n]$ che sommati restituiscono un dato valore intero K. Tale problema può essere risolto utilizzando la programmazione dinamica, considerando i seguenti problemi $P(i,j)$, con $i \in \{1,\dots,n\}$ e $j \in \{0,\dots,K\}$, così definiti:

    $P(i,j) = true$ se esiste un sottoinsieme di elementi presi tra i primi i elementi dell'array A che sommati danno j, altrimenti $P(i,j) = false$.

    I problemi $P(i,j)$ possono essere risolti in modo iterativo rispetto all'indice i tenendo in considerazione che:

$$P(i,j) = \begin{cases} true & \text{se } j = 0 \\ true & \text{se } j > 0,\ i = 1 \text{ e } A[1] = j \\ false & \text{se } j > 0,\ i = 1 \text{ e } A[1] \neq j \\ P(i-1,j) & \text{se } j > 0,\ i > 1 \text{ e } A[i] > j \\ P(i-1,j) \text{ or } P(i-1,j-A[i]) & \text{se } j > 0,\ i > 1 \text{ e } A[i] \leq j \end{cases}$$

    La soluzione al problema iniziale coincide con $P(n,K)$, ovvero il sottoproblema che considera tutti i valori in ingresso e la somma richiesta risulta essere K.

    L'Algoritmo 37 risolve tutti i problemi $P(i,j)$ salvando le relative soluzioni in una matrice booleana B, e alla fine restituisce $B[n,K]$. Il costo computazionale risulta essere $\Theta(n \times K)$ in quanto vengono eseguite alcune operazioni di costo costante per ogni cella della matrice B, avente dimensione $n \times (K+1)$, ma $\Theta(n \times (K+1)) = \Theta(n \times K) + \Theta(n) = \Theta(n \times K)$.

Algorithm 37: PIETRE(INT $A[1..n]$, INT K) → BOOLEAN

```
BOOLEAN B[1..n, 0..K]
for i ← 1 to n do
    B[i,0] ← true
for j ← 1 to K do
    if A[1] = j then
        B[1,j] ← true
    else
        B[1,j] ← false
for i ← 2 to n do
    for j ← 1 to K do
        if A[i] > j then
            B[i,j] ← B[i-1,j]
        else
            B[i,j] ← B[i-1,j] or B[i-1,j-A[i]]
return B[n,K]
```

Cont.

11. **Soluzione.** È possibile utilizzare la programmazione dinamica considerando i problemi $T(i,j)$, con $i \in \{1,\ldots,n\}$ e $j \in \{1,\ldots,K\}$, tale che $T(i,j)$ = massimo peso raggiungibile utilizzando un sottoinsieme dei pacchi con indici in $\{1,\ldots,i\}$ aventi un valore complessivo inferiore o uguale a j. I problemi $T(i,j)$ possono essere risolti considerando che:

$$T(i,j) = \begin{cases} 0 & \text{se } i = 1 \text{ e } j < v_1 \\ p_1 & \text{se } i = 1 \text{ e } j \geq v_1 \\ T(i-1,j) & \text{se } i > 1 \text{ e } j < v_i \\ max\{T(i-1,j), T(i-1,j-v_i) + p_i\} & \text{se } i > 1 \text{ e } j \geq v_i \end{cases}$$

L'Algoritmo 38 risolve i problemi $T(i,j)$ inserendo le relative soluzioni nella tabella $T[1..n, 1..W]$. L'esercizio richiede di stampare l'insieme di pacchi ottimale (e non il loro valore memorizzato in $T[n,K]$); a tal fine l'algoritmo utilizza una matrice di booleani $B[i,j]$ che indica se l'oggetto i-esimo fa parte della soluzione al problema $T(i,j)$. Abbiamo che il costo computazionale di tale algoritmo risulta essere $T(n,K) = \Theta(n \times K)$ in quanto le operazioni elementari hanno tutte costo costante, ed il blocco di maggiore costo è quello che contiene i due **for** annidati le cui operazioni interne vengono eseguite $(n-1) \times K$ volte.

Algorithm 38: TRASPORTATORE(INT $P[1..n]$, INT $V[1..n]$, INT K)

```
// Inizializzazione prima riga delle tabelle T e B
for j ← 1 to W do
    if V[1] ≤ j then
        T[1,j] ← P[1]; B[1,j] ← true
    else
        T[1,j] ← 0; B[1,j] ← false
// Riempimento restanti righe delle tabelle T e B
for i ← 2 to n do
    for j ← 1 to W do
        if V[i] ≤ j and T[i−1,j] < T[i−1,j−V[i]] + P[i] then
            T[i,j] ← T[i−1,j−V[i]] + P[i]; B[i,j] ← true        // si prende l'oggetto i-esimo
        else
            T[i,j] ← T[i−1,j]; B[i,j] ← false                  // non si prende l'oggetto i-esimo
if T[n,K] ≠ 0 then
    // Esiste una soluzione e viene stampata
    pacco ← n; valoreRimanente ← K
    while pacco > 0 and valoreRimanente > 0 do
        if B[pacco, valoreRimanente] then
            stampa(pacco)
            valoreRimanente ← valoreRimanente − V[pacco]
        pacco ← pacco − 1
```

Cont.

12. **Soluzione.** Il problema corrisponde con la ricerca del $k+1$-esimo minimo. Possiamo utilizzare l'algoritmo quickselect.

Algorithm 39: CercaKmenoUnesimoMinimo(Number $v[1..n]$,Int k) $\rightarrow$ Number

```
s ← 1; e ← n        // s ed e delimitano il sottovettore su cui si sta effettuando la ricerca
while true do
    scegli casualmente un valore v[w] con w compreso tra s e e
    (iniziox, finex) ← Partiziona(v[1..n], s, e, v[w])
    if iniziox ≤ k + 1 ≤ finex then
        return v[w]                                  // l'elemento da cercare coincide con v[w]
    else if k + 1 < iniziox then
        e ← iniziox − 1                              // l'elemento da cercare è minore di v[w]
    else
        s ← finex + 1                                // l'elemento da cercare è maggiore di v[w]
```

Algorithm Partiziona(Int v, Int *start*, Int *end*, Number x) $\rightarrow$ (Int, Int)

```
/* bandiera nazionale:  minori di x verdi, uguali a x bianchi, maggiori di x rossi    */
/* restituisce una coppia di indici:  inizio e fine della collocazione dei valori x   */
i ← start; j ← start; k ← end
while j < k do
    if A[j] < x then
        if i < j then
            scambia A[i] con A[j]
        i ← i + 1; j ← j + 1
    else if A[j] > x then
        scambia A[j] con A[k]
        k ← k − 1
    else
        j ← j + 1                                    // in questo caso A[j] = x
return (i, j − 1)          // i valori x (bianchi) sono dalla posizione i alla posizione j − 1
```

L'algoritmo CercaKmenoUnesimoMinimo (si veda Algoritmo 39) è una versione iterativa di quickselect che ha quindi il medesimo costo computazionale studiato a lezione: risulta quindi essere lineare nel caso ottimo e medio, quadratico nel caso pessimo.

Cont.

13. **Soluzione.** È possibile utilizzare la programmazione dinamica considerando i problemi $P(i,j)$, con indici $i \in \{1,\ldots,n\}$ e $j \in \{0,\ldots,H\}$, tale che $P(i,j)$ = massimo numero di vaccinazioni considerando le scuole con indici in $\{1,\ldots,i\}$ e un tempo a disposizione dell'infermiere pari a j. I problemi $P(i,j)$ possono essere risolti considerando che:

$$P(i,j) = \begin{cases} 0 & \text{se } i = 1 \text{ e } j < h_1 \\ b_1 & \text{se } i = 1 \text{ e } j \geq h_1 \\ P(i-1,j) & \text{se } i > 1 \text{ e } j < h_i \\ max\{P(i-1,j), P(i-1,j-h_i)+b_i\} & \text{se } i > 1 \text{ e } j \geq h_i \end{cases}$$

L'Algoritmo 40 risolve i problemi $P(i,j)$ inserendo le relative soluzioni nella tabella $P[1..n, 0..H]$. L'esercizio richiede di stampare le scuole che permettono di ottenere un numero massimo di vaccinazioni complessive (e non il totale delle vaccinazioni memorizzato in $P[n,K]$); a tal fine l'algoritmo utilizza una matrice di booleani $B[i,j]$ che indica se la scuola i-esima fa parte della soluzione al problema $P(i,j)$. Abbiamo che il costo computazionale di tale algoritmo risulta essere $T(n,K) = \Theta(n \times K)$ in quanto le operazioni elementari hanno tutte costo costante, ed il blocco di maggiore costo è quello che contiene i due **for** annidati le cui operazioni interne vengono eseguite $(n-1) \times K$ volte.

Algorithm 40: VACCINAZIONI(INT H, INT $b[1..n]$, INT $h[1..n]$,)

```
// Inizializzazione prima riga delle tabelle P e B
for j ← 0 to H do
    if j < h[1] then
        P[1,j] ← 0; B[1,j] ← false
    else
        P[1,j] ← b[1]; B[1,j] ← true
// Riempimento restanti righe delle tabelle P e B
for i ← 2 to n do
    for j ← 0 to H do
        if j ≥ h[i] and P[i−1,j] < P[i−1,j−h[i]] + b[i] then
            P[i,j] ← P[i−1,j−h[i]] + b[i]; B[i,j] ← true        // si sceglie la scuola i-esima
        else
            P[i,j] ← P[i−1,j]; B[i,j] ← false                  // non si sceglie la scuola i-esima
if P[n,H] ≠ 0 then
    // Si possono effettuare vaccinazioni e si stampano le scuole dove andare
    scuola ← n; oreRimanenti ← H
    while scuola > 0 do
        if B[scuola, oreRimanenti] then
            stampa(scuola)
            oreRimanenti ← oreRimanenti − h[scuola]
        scuola ← scuola − 1
```

Cont.

14. **Soluzione** È possibile procedere utilizzando due ricerche binarie: la prima ricerca considera le colonne e identifica la possibile colonna che potrebbe contenere x, mentre la seconda ricerca considera tale colonna e verifica se il valore x appare all'interno della colonna.

    L'Algoritmo 41 effettua tali ricerche in modo iterativo utilizzando i seguenti indici: $colStart$ e $colEnd$ che indicano l'intervallo di colonne di interesse durante la prima ricerca, e $rowStart$ e $rowEnd$ che indicano l'intervallo di righe di interesse durante la seconda ricerca. Si noti che prima di inizializzare e aggiornare tali indici, si verifica sempre che valga l'invariante $T[rowStart, colStart] \leq x \leq T[rowEnd, colEnd]$. Nel caso in cui non sia possibile mantenere l'invariante, si restituisce immediatamente $false$ in quanto x non appare nella tabella.

    Utilizzando ricerche binarie, gli intervalli vengono dimezzati ad ogni iterazione. Abbiamo quindi che nel caso pessimo, che si verifica quando x appare in T, serviranno $\Theta(\log m)$ iterazioni durante la ricerca della colonna e $\Theta(\log n)$ iterazioni durante la ricerca della riga. Nel caso pessimo avremo quindi $T(n, m) = \Theta(\log m) + \Theta(\log n) = \Theta(\log(max(n, m)))$. Nel caso ottimo, che occorre quando $x < T[1, 1]$ oppure $x > T[n, m]$ e quindi si restituisce immediatamente $false$, il costo è costante, ovvero $T(n, m) = \Theta(1)$.

Algorithm 41: RICERCA(NUMBER $T[1..n, 1..m]$, NUMBER x) $\rightarrow$ BOOLEAN

```
if x < T[1,1] or x > T[n,m] then
    return false
INT colStart ← 1, colEnd ← m, colAvg
INT rowStart ← 1, rowEnd ← n, rowAvg
// Invariante: T[rowStart, colStart] ≤ x ≤ T[rowEnd, colEnd]
while colStart < colEnd do
    colAvg ← ⌊(colStart + colEnd)/2⌋
    if x ≤ T[rowEnd, colAvg] then
        colEnd ← colAvg
    else
        if T[rowStart, colAvg + 1] ≤ x then
            colStart ← colAvg + 1
        else
            return false
// A questo punto vale colStart = colEnd
while rowStart < rowEnd do
    rowAvg ← ⌊(rowStart + rowEnd)/2⌋
    if x ≤ T[rowAvg, colStart] then
        rowEnd ← rowAvg
    else
        if T[rowAvg + 1, colStart] ≤ x then
            rowStart ← rowAvg + 1
        else
            return false
// A questo punto vale rowStart = rowEnd e dall'invariante T[rowStart, colStart] = x
return true
```

Cont.

5.4 Esercizi su Grafi

1. **Soluzione.** È possibile usare una visita in ampiezza partendo dal vertice v. Infatti, in questo modo si visitano gli archi in ordine di distanza non decrescente a partire da v. Appena si visita un nodo adiacente a v, si può concludere che la lunghezza del ciclo minimo coincide con la distanza di tale nodo più 1. Se si termina la visita senza trovare alcun ciclo, tale ciclo non esiste e si può restituire ∞.

Algorithm 42: CicloMinimo(Graph $G = (V, E)$, Vertex v) $\rightarrow$ Int

```
Queue q ← new Queue()
for x ∈ V do
    x.mark ← false
    x.dist ← ∞
v.mark ← true
v.dist ← 0
q.enqueue(v)
while not q.isEmpty() do
    u ← q.dequeue()
    if v ∈ u.adjacents() then
        return u.dist + 1
    else
        for w ∈ u.adjacents() do
            if not w.mark then
                w.mark ← true
                w.dist ← u.dist + 1
                q.enqueue(w)
return ∞
```

Il costo computazionale dell'algoritmo coincide con quello della visita in ampiezza, ovvero $O(m + n)$ con m numero di archi ed n numero di vertici del grafo. Si noti che a differenza della visita in ampiezza, nel caso in cui esista il ciclo, tale algoritmo può terminare prima di aver visitato l'intero grafo. Ma nel caso pessimo, che si verifica in assenza di tale ciclo, si rende necessario visitare l'intero grafo.

Cont.

2. **Soluzione.** È possibile usare una versione modificata dell'algoritmo di Dijkstra che utilizza una coda con priorità invertita, cioè che restituisce il valore con priorità massima e non il valore con priorità minima. Al termine dell'esecuzione dell'algoritmo di Dijkstra, si stampano i nodi del cammino da s a t nell'albero dei cammini minimi memorizzata nel parent-vector *pred*. Le strutture dati usate sono le solite dell'algoritmo di Dijkstra: il vettore D delle distanze e la coda con priorità (massima) Q.

Algorithm 43: CamminoMassimo(Grafo $G = (V, E, w)$, Int s, Int t)

```
/* inizializzazione strutture dati                                          */
n ← G.numNodi()
Int pred[1..n], v, u
Double D[1..n]
for i ← 1 to n do
    D[i] ← −∞
    pred[i] ← −1
D[s] ← 0
MaxPriorityQueue[Int, Double] Q ← new MaxPriorityQueue[Int, Double]()
Q.insert(s, D[s])

/* esecuzione algoritmo di Dijkstra                                         */
while not Q.isEmpty() do
    u ← Q.findMax()
    Q.deleteMax()
    for v ∈ u.adjacent() do
        if D[v] = −∞ then
            /* prima volta che si incontra v                                */
            D[v] ← D[u] + w(u, v)
            Q.insert(v, D[v])
            pred[v] = u
        else if D[u] + w(u, v) > D[v] then
            /* scoperta di un cammino migliore per raggiungere v            */
            Q.increaseKey(v, −D[v] + D[u] + w(u, v))
            D[v] = D[u] + w(u, v)
            pred[v] = u
/* stampa del cammino da s a t                                              */
printPath(pred, s, t)

/* funzione ausiliaria di stampa del cammino                                */
procedure printPath( Int pred[], Int s, Int t)
if pred[t] ≠ −1 then
    printPath(pred, s, pred[t])
    print t
```

Il costo computazionale dell'algoritmo è la medesima dell'algoritmo di Dijkstra, ovvero $T(n, m) = O(m \log n)$, dove n è il numero di vertici ed m il numero di archi nel grafo.

Cont.

3. **Soluzione.** Il problema prevede di trovare il vertice appartenente ad R che ha il cammino minore per raggiungere il vertice p. Essendo il grafo non orientato, questo coincide con il vertice appartenente ad R a distanza minima da p. I pesi saranno non negativi in quanto quantificano degli intervalli di tempo, quindi è possibile utilizzare l'algoritmo di Dijkstra.

   L'algoritmo ANNAFFIA è una versione dell'algoritmo di Dijkstra che visita i nodi in ordine di distanza non decrescente da p, e che interrompe l'esecuzione appena si raggiunge un nodo appartenente ad R. Se si termina l'esecuzione dell'algoritmo di Dijkstra senza raggiungere nodi appartenenti ad R, allora non è possibile annaffiare la pianta p e si restituisce un errore. In questo modo, il costo computazionale dell'algoritmo ANNAFFIA nel caso pessimo coincide con il costo dell'algoritmo di Dijkstra, ovvero $O(m \log n)$ dove $m = |E|$ e $n = |V|$.

Algorithm 44: ANNAFFIA(GRAFO $G = (V, E, w)$, SET[VERTEX] R, VERTEX p) $\rightarrow$ VERTEX

```
/* inizializzazione strutture dati                                          */
n ← G.numNodi()
DOUBLE D[1..n]
for i ← 1 to n do
    D[i] ← ∞
D[p] ← 0
MINPRIORITYQUEUE[INT, DOUBLE] Q ← new MINPRIORITYQUEUE[INT, DOUBLE]()
Q.insert(p, D[p])

/* esecuzione algoritmo di Dijkstra                                         */
while not Q.isEmpty() do
    u ← Q.findMin()
    if u ∈ R then
        return u
    Q.deleteMin()
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            /* prima volta che si incontra v                                */
            D[v] ← D[u] + w(u, v)
            Q.insert(v, D[v])
        else if D[u] + w(u, v) < D[v] then
            /* scoperta di un cammino migliore per raggiungere v            */
            Q.decreaseKey(v, D[v] − D[u] − w(u, v))
            D[v] = D[u] + w(u, v)
return error
```

Cont.

4. **Soluzione.** Considerando che dobbiamo confrontare tanti cammini minimi, risulta conveniente l'utilizzo dell'algoritmo di Floyd-Warshall per il calcolo di tutti i cammini minimi tra tutte le coppie di vertici di un grafo orientato pesato. Una volta calcolati tutti i cammini minimi, si controllano tutti i cammini fra coppie di nodi (v_1, v_2) con $v_1 \in V_1$ e $v_2 \in V_2$, per scegliere il più piccolo fra tutti questi. Una volta trovata la coppia (v_1, v_2) con il cammino minore, si procede a stampare il relativo cammino.

Algorithm 45: MinimoCamminoMinimo(Grafo $G = (V, E, w)$, Set[Vertex] $V1$, Set[Vertex] $V2$,)

```
n ← G.numNodi()                                   // esecuzione dell'algoritmo di Floyd-Warshall
Real D[1..n, 1..n]
Int next[1..n, 1..n]
for x ← 1 to n do
    for y ← 1 to n do
        if x = y then
            D[x, y] = 0
            next[x, y] = −1
        else if (x, y) ∈ E then
            D[x, y] = w(x, y)
            next[x, y] = −1
        else
            D[x, y] = ∞
            next[x, y] = −1
for k ← 1 to n do
    for x ← 1 to n do
        for y ← 1 to n do
            if D[x, k] + D[k, y] < D[x, y] then
                D[x, y] = D[x, k] + D[k, y]
                next[x, y] = next[x, k]
Int v1min, v2min, min ← ∞          // ricerca miglior cammino da un nodo in V1 a un nodo in V2
for v1 ∈ V1 do
    for v2 ∈ V2 do
        if D[v1, v2] < min then
            min ← D[v1, v2]
            v1min ← v1
            v2min ← v2
PrintPath(v1min, v2min, next)                      // stampa del cammino minimo da v1min a v2min

function PrintPath(Int u, v, next[1..n, 1..n])  // stampa cammino per l'algoritmo Floyd-Warshall
if ¬(u = v) ∧ (next[u, v] < 0) then
    errore(u e v non sono connessi)
else
    print u
    while ¬(u = v) do
        u ← next[u, v]
        print u
```

Il costo computazionale dell'algoritmo è il medesimo dell'algoritmo di Floyd-Warshall, ovvero $T(n, m) = O(n^3)$, dove n è il numero di vertici ed m il numero di archi nel grafo. Si noti infatti che le parti aggiuntive dell'algoritmo servono per ricercare i vertici $v1min$ di inizio e $v2min$ di fine del cammino minimo (con costo $O(n^2)$) e per stampare il cammino da $v1min$ a $v2min$ (con costo $O(n)$). Tali costi aggiuntivi sono inferiori in ordine di grandezza e vengono quindi assorbiti dal costo dell'algoritmo di Floyd-Warshall.

Cont.

5. **Soluzione.** È sufficiente effettuare una visita in ampiezza del grafo a partire da v e restituire la distanza dell'ultimo nodo visitato, visto che la BFS visita i vertici in ordine non decrescente di distanza dal vertice di inizio della visita. L'algoritmo (si veda Algoritmo 46) ha il medesimo costo computazionale della visita

Algorithm 46: RAGGIO(GRAPH $G = (V, E)$, VERTEX v) $\rightarrow$ INT

```
QUEUE q ← new QUEUE()
for x ∈ V do
    x.mark ← false
    x.dist ← ∞
v.mark ← true
v.dist ← 0
q.enqueue(v)
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if not w.mark then
            w.mark ← true
            w.dist ← u.dist + 1
            q.enqueue(w)
return u.dist                                   // u contiene l'ultimo vertice visitato
```

in ampiezza che, assumendo implementazione tramite liste di adiacenza, e sapendo che il grafo è connesso (quindi con un numero di archi maggiore o uguale, in ordine di grandezza, del numero di vertici), risulta essere $\Theta(|E|)$, in quanto tutti gli archi vengono controllati almeno una volta.

Cont.

6. **Soluzione.** Tra gli algoritmi visti a lezione, è possibile utilizzare Floyd-Warshall e al termine della sua esecuzione controllare se è stata calcolata una distanza negativa fra un vertice e se stesso. Tale soluzione (si veda Algoritmo 47) ha un costo computazionale coincidente con il costo dell'algoritmo di Floyd-Warshall, ovvero $\Theta(|V|^3)$.

   Si noti che non è possibile utilizzare direttamente l'algoritmo di Bellman-Ford in quanto tale algoritmo considera un dato vertice di partenza ed è in grado di verificare la presenza di cicli di costo negativo raggiungibili da tale vertice. Quindi, se si esegue Bellman-Ford a partire da un vertice $v \in V$, ed esistono cicli di costo negativo che coinvolgono solo vertici non raggiungibili da v, tale algoritmo non sarà in grado di verificare la presenza di tali cicli.

Algorithm 47: VERIFICACICLINEGATIVI(GRAFO $G = (V, E, w) \rightarrow$ BOOL

```
// esecuzione dell'algoritmo di Floyd-Warshall
n ← G.numNodi()
REAL D[1..n, 1..n]
for x ← 1 to n do
    for y ← 1 to n do
        if x = y then
            D[x, y] = 0
        else if (x, y) ∈ E then
            D[x, y] = w(x, y)
        else
            D[x, y] = ∞
for k ← 1 to n do
    for x ← 1 to n do
        for y ← 1 to n do
            if D[x, k] + D[k, y] < D[x, y] then
                D[x, y] = D[x, k] + D[k, y]
// verifica presenza cicli negativi
for v ∈ V do
    if D[v, v] < 0 then
        return true
return false
```

Cont.

7. **Soluzione.** Il problema può essere interpretato come un problema su grafi, dove le stanze rappresentano i vertici, le gallerie gli archi, ed i tempi di costruzione delle gallerie i pesi degli archi. Abbiamo in input un grafo completo (quindi con un numero di archi quadratico rispetto al numero di vertici), in quanto esiste una potenziale galleria per ogni coppia di stanze. Le gallerie che minimizzano i tempi di costruzione coincidono con un minimum spanning tree per tale grafo. Una volta trovato il tempo complessivo necessario per costruire le gallerie del minimum spanning tree, si aggiunge il tempo di costruzione delle stanze. L'Algoritmo 48 utilizza l'algoritmo di Prim prendendo in input l'insieme S, e due strutture K e T che consideriamo essere rispettivamente un array ed una tabella che utilizza gli elementi di S come indici (a titolo di esempio, usiamo $T[s, v]$, con $s, v \in S$, per indicare il tempo di costruzione della galleria dalla stanza s alla stanza v). Anche la struttura dati ausiliaria $D[S]$ è un array indicizzato su S. Questa è una pseudo-notazione di comodo: una vera implementazione potrebbe usare la convenzione che le stanze vengono rappresentate dall'insieme di indici $\{1, \ldots, |S|\}$. Durante il calcolo del minimum spanning tree, ad ogni selezione di una galleria, si incrementa un contatore *tot* con il relativo tempo di costruzione. Al termine del calcolo del minimum spanning tree, si aggiungono a *tot* i tempi di costruzione delle stanze. Infine, si restituisce il contatore *tot*. Il costo computazionale corrisponde con il costo dell'algoritmo di Prim che, considerando il grafo completo, può essere espresso come $T(n) = O(n^2 \log n)$, dove $n = |S|$ è il numero delle stanze.

Algorithm 48: FORMICAIO(SET S, NUMBER $K[S]$, NUMBER $T[S, S]$) $\rightarrow$ NUMBER

```
/* inizializzazione strutture dati                                    */
INT v, u
NUMBER D[S], tot
SET visited
MINPRIORITYQUEUE[S, NUMBER] Q ← new MINPRIORITYQUEUE[S, NUMBER]()
s ← S.selectOneElement()
D[s] ← 0; Q.insert(s, D[s]); visited ← {s}
for v ∈ S \ {s} do
    D[v] ← T[s, v]; Q.insert(v, D[v])

/* esecuzione algoritmo di Prim (modificato)                          */
tot ← 0
while not Q.isEmpty() do
    u ← Q.findMin(); Q.deleteMin(); visited ← visited ∪ {u}
    tot ← tot + D[u]
    for v ∈ S \ visited do
        if T[u, v] < D[v] then
            /* scoperta di una galleria migliore per raggiungere v    */
            Q.decreaseKey(v, D[v] − T[u, v])
            D[v] = T[u, v]

/* aggiunta dei tempi di costruzione delle stanze                     */
for v ∈ S do
    tot ← tot + K[v]

return tot
```

Cont.

8. **Soluzione.** Per cercare il cammino minimo fra due vertici dati è possibile utilizzare una versione modificata dell'algoritmo di Dijkstra in quanto i costi degli archi, e dei pedaggi, risultano essere non negativi. La modifica rispetto all'algoritmo tradizionale deriva dal fatto che, nel momento del calcolo della distanza di un vertice v, si deve prendere in considerazione anche il costo $t(v)$ del pedaggio per entrare nella relativa città.

   L'Algoritmo 49 utilizza la solita convenzione secondo cui i vertici vengono rappresentati da numeri interi nell'intervallo $\{1 \dots |V|\}$; in questo modo la funzione $t : V \to \mathbb{R}$ viene rappresentata tramite un array di $|V|$ numeri (un numero di costo del pedaggio per ogni vertice). L'algoritmo ha il medesimo costo computazionale dell'algoritmo di Dijkstra, quindi $T(n, m) = O(m \log n)$ con $n = |V|$ e $m = |E|$.

Algorithm 49: CamminoPiuEconomico(Grafo $G = (V, E, w)$, Number $t[1..|V|]$, Int s, Int d)

```
// inizializzazione strutture dati
n ← G.numNodi()
Int pred[1..n], v, u
Number D[1..n]
for i ← 1 to n do
    D[i] ← ∞
    pred[i] ← −1
D[s] ← 0
MinPriorityQueue[Int, Number] Q ← new MinPriorityQueue[Int, Number]()
Q.insert(s, D[s])

// esecuzione algoritmo di Dijkstra (modificato)
while not Q.isEmpty() do
    u ← Q.findMin()
    Q.deleteMin()
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            // prima volta che si incontra v
            D[v] ← D[u] + w(u, v) + t[v]
            Q.insert(v, D[v])
            pred[v] = u
        else if D[u] + w(u, v) + t[v] < D[v] then
            // scoperta di un cammino migliore per raggiungere v
            Q.decreaseKey(v, D[v] − D[u] − w(u, v) − t[v])
            D[v] = D[u] + w(u, v) + t[v]
            pred[v] = u
// stampa del cammino da s a d
printPath(pred, d)

// funzione ausiliaria di stampa del cammino
procedure printPath(Int pred[1..|V|], Int d)
if pred[d] ≠ −1 then
    printPath(pred, pred[d])
    print d
```

Cont.

9. **Soluzione.** Il problema proposto prevede una stampa dei vertici del grafo inversa rispetto ad una stampa secondo un ordinamento topologico. Un ordinamento topologico può essere ottenuto eseguendo una visita in profondità a considerando i vertici in ordine inverso di "chiusura" (ovvero il momento in cui vengono marcati come "black" secondo l'algoritmo standard che usa le colorazioni dei vertici "white", "gray" e "black"). Il problema può quindi essere risolto tramite una visita in profondità che stampa i vertici nel momento di chiusura della loro visita. Tale soluzione è riportata come Algoritmo 50 e ha il medesimo costo della visita in profondità, ovvero $O(n+m)$, con n numero dei vertici e m numero degli archi, assumendo implementazione del grafo tramite liste di adiacenza.

Algorithm 50: InversoVisitaTopologica(Graph (E, V))

```
// Inizializzazione marcatura dei vertici
for v ∈ V do
    v.mark ← white
// Esecuzione della DFS
for v ∈ V do
    if v.mark = white then
        DFSvisit(v)

DFSvisit(Vertex u)
u.mark ← gray
for v ∈ u.adjacents do
    if v.mark = white then
        DFSvisit(v)
// Si stampa u in quanto tutti i nodi raggiungibili da u sono già stati visitati
print(u)
```

Cont.

10. **Soluzione.** È possibile procedere effettuando una visita in profondità, semplicemente verificando l'esistenza di almeno un arco all'indietro. A lezione si era discusso del fatto che questo si verifica se e solo se il grafo visitato contiene un ciclo. Più precisamente, si esegue una DFS che restituisce *true* se e solo se durante l'esecuzione della visita si incontra un arco che va da un vertice attualmente visitato a un vertice "grigio", ovvero già visitato ma non ancora chiuso. Tale soluzione è riportata come Algoritmo 51 e ha il medesimo costo della visita in profondità, ovvero $O(n+m)$, con n numero dei vertici e m numero degli archi, assumendo implementazione del grafo tramite liste di adiacenza.

Algorithm 51: ControllaCiclo(Graph (E, V)) $\rightarrow$ Boolean

```
// Inizializzazione marcatura dei vertici
for v ∈ V do
    v.mark ← white
// Esecuzione della DFS
for v ∈ V do
    if v.mark = white then
        if DFSvisit(v) then
            return true
// Se la DFS non ha incontrato archi all'indietro non vi sono cicli
return false

// DFS che restituisce true se e solo se si incontra un arco all'indietro
DFSvisit(Vertex u) → Boolean
u.mark ← gray
for v ∈ u.adjacents do
    if v.mark = white then
        if DFSvisit(v) then
            return true
    else if v.mark = gray then
        return true
// La DFS di u non ha trovato archi all'indietro
u.mark ← black
return false
```

Cont.

11. **Soluzione.** Ciò che viene richiesto può essere ottenuto con una stampa dei vertici del grafo secondo un ordinamento topologico. Infatti, l'esistenza di un cammino da v_i a v_j implica l'esistenza di una sequenza di archi (v_i, v_{i+1}), (v_{i+1}, v_{i+2}), $\ldots$, (v_{i+l}, v_j) che implica che, in un ordinamento topologico, v_i apparirà prima di v_j. Come descritto a lezione, un ordinamento topologico può essere ottenuto considerando i vertici in ordine inverso di chiusura secondo una visita in profondità.

    L'Algoritmo 52 effettua una visita DFS del grafo che inserisce i vertici visitati in una struttura LIFO, al momento della chiusura della loro visita. Al termine della visita, i vertici vengono stampati secondo l'ordine di estrazione da tale struttura LIFO. Il costo computazionale di tale algoritmo è il medesimo della visita DFS, quindi $O(n + m)$ (con n numero di nodi e m numero di vertici, assumendo implementazione tramite liste di adiacenza) in quanto le operazioni che sono state aggiunte all'algoritmo DFS sono, per ogni vertice, un inserimento, una lettura ed una cancellazione dallo stack. Queste operazioni contribuiscono al costo computazionale con una quantità $O(n)$, che è trascurabile rispetto al costo $O(n + m)$ della DFS.

Algorithm 52: OrdineTopologico(Graph (E, V))

```
// Inizializzazione stack e marcatura dei vertici
Stack s ← new Stack ()
for v ∈ V do
    v.visited ← false
// Esecuzione della DFS
for v ∈ V do
    if not v.visited then
        DFSvisit(v)
// Estrazione dei vertici dallo stack e stampa
while not s.isEmpty() do
    print s.top()
    s.pop()

// DFS che inserisce i vertici nello stack al momento della chiusura
DFSvisit(Vertex u)
u.visited ← true
for v ∈ u.adjacents do
    if not v.visited then
        DFSvisit(v)
s.push(u)
```

Cont.

12. **Soluzione.** Visto che l'algoritmo di visita in ampiezza (BFS) visita i vertici in ordine di distanza dal vertice di partenza della visita, è sufficiente effettuare una BFS a partire da v_1 e stampare il cammino che viene trovato per raggiungere v_2.

Algorithm 53: CamminoLunghezzaMinima(Graph $G = (V, E)$, Vertex v_1, Vertex v_2)

```
Queue q ← new Queue()
for x ∈ V do
    x.mark ← false
    x.parent ← −1
v1.mark ← true
q.enqueue(v1)
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if w = v2 then
            v2.parent = u
            Stampa(v1, v2)
        if not w.mark then
            w.mark ← true
            w.parent ← u
            q.enqueue(w)
```

Algorithm Stampa(Vertex s, Vertex e)

```
if s=e then
    print(s)
else
    Stampa(s, e.parent)
    print(e)
```

L'algoritmo (si veda Algoritmo 53) ha il medesimo costo computazionale della visita in ampiezza che, assumendo implementazione tramite liste di adiacenza, e sapendo che il grafo è connesso (quindi con un numero di archi maggiore o uguale, in ordine di grandezza, del numero di vertici), risulta essere $O(|E|)$. Infatti, nel caso pessimo, l'algoritmo visita tutti i vertici e quindi deve considerare tutti gli archi.

Cont.

13. **Soluzione.** Non sapendo se esistono o meno archi di peso negativo, per trovare cammini di costo minimo si adotta l'algoritmo di Bellman-Ford. Per trovare un ciclo di costo minimo che contiene v, si può eseguire l'algoritmo di Bellman-Ford considerando v come vertice iniziale. Una volta eseguito l'algoritmo, per ogni vertice u avremo in $D[u]$ il costo minimo dei cammini da v ad u. Si cerca quindi il vertice k, per cui esiste un arco (k, v), che minimizza $D[k]+w(k, v)$. Se esiste un tale vertice k, abbiamo che un ciclo di costo minimo che contiene v può essere ottenuto considerando un cammino di costo minimo da v ad k, aggiungendo in coda a tale cammino il vertice di partenza v. L'Algoritmo 54 descrive in pseudocodice tale soluzione al problema. La parte iniziale corrisponde all'algoritmo di Bellman-Ford e ha costo computazionale $\Theta(n \times m)$ con n numero dei vertici e m numero degli archi del grafo. A seguire sono presenti il tipico controllo di possibili cicli negativi associato all'algoritmo di Bellman-Ford (costo $\Theta(m)$), la ricerca del penultimo vertice k del ciclo (costo $\Theta(n)$) e la eventuale stampa del ciclo (costo $O(n)$). Sommando tutti questi contributi avremo che il costo computazionale della soluzione proposta risulta essere $T(n) = \Theta(n \times m) + \Theta(m) + \Theta(n) + O(n) = \Theta(n \times m)$.

Algorithm 54: CicloCostoMinimo(Graph G=(V, E, w), Vertex v)

```
// esecuzione dell'algoritmo di Bellman-Ford
n ← G.numNodi()
Real D[1..n]; Int pred[1..n];
for each u ∈ {1...n} do
    D[u] = ∞
    pred[u] = −1
D[v] ← 0
for i ← 1 to n − 1 do
    for each (u, z) ∈ E do
        if D[u] + w(u, z) < D[z] then
            D[z] = D[u] + w(u, z)
            pred[z] = u
// verifica presenza cicli costo negativo
for each (u, z) ∈ E do
    if D[u] + w(u, z) < D[z] then
        error("il grafo contiene cicli negativi")
// ricerca del vertice k tale che v è adiacente a k e D[k] + w(k, v) è minimo
Vertex k; Real min ← ∞
for each u ∈ {1...n} do
    if (u, v) ∈ E and D[u] + w(u, v) < min then
        k ← u
        min ← D[u] + w(u, v)
// stampa ciclo di costo minimo
if min = ∞ then
    stampa("no ciclo minimo")
else
    PrintPath(pred, k)
    print(v)

function PrintPath(Int[1..n] pred, Int t)
if pred[t] ≠ −1 then
    PrintPath(pred, pred[t])
print(t)
```

Cont.

14. **Soluzione** Innanzitutto assumiamo che il grafo che rappresenta il campo da gioco sia connesso, altrimenti lo scopo del gioco sarebbe banalmente non raggiungibile. In tal caso, verificare la raggiungibilità dello scopo del gioco, coincide con la verifica dell'esistenza di un albero di copertura con costo complessivo inferiore a K. Infatti, per far in modo che il passaparola raggiunga tutti i punti del campo di gioco, è necessario che vengano percorsi dai bambini dei cammini, corrispondenti ad archi che ricoprono l'intero grafo. Inoltre, il numero complessivo di caramelle necessarie coinciderà con la somma dei pesi degli archi associati ai cammini effettuati.

    L'Algoritmo 55 innanzitutto calcola il costo del minimum spanning tree utilizzando l'algoritmo di Kruskal. Tale costo viene memorizzato nella variabile *tot*. Successivamente, l'algoritmo controlla semplicemente se $tot \leq K$; in tal caso lo scopo del gioco è raggiungibile, altrimenti non lo è. Il costo computazionale dell'algoritmo corrisponde con quello dell'algoritmo di Kruskal, ovvero $T(n, m) = O(m \log n)$ con $n = |V|$ e $m = |E|$.

Algorithm 55: PASSAPAROLA(GRAPH $G = (V, E, w)$, VERTEX s, INT K) $\rightarrow$ BOOLEAN

```
INT tot ← 0
UNIONFIND UF
for each v ∈ V do
    UF.makeSet(v)
SORT(E, w)
for each {u, v} ∈ E do
    Tu ← UF.find(u)
    Tv ← UF.find(v)
    if Tu ≠ Tv then
        tot ← tot + w(u, v)
        UF.union(Tu, Tv)
if tot ≤ K then
    return true
else
    return false
```